Cooperating Proof Attempts

Giles Reger, Dmitry Tishkovsky, and Andrei Voronkov [⋆]

University of Manchester, Manchester, UK

Abstract. This paper introduces a pseudo-concurrent architecture for first-order saturation-based theorem provers with the eventual aim of developing it into a truly concurrent architecture. The motivation behind this architecture is two-fold. Firstly, first-order theorem provers have many configuration parameters and commonly utilise multiple strategies to solve problems. It is also common that one of these strategies will solve the problem quickly but it may have to wait for many other strategies to be tried first. The architecture we propose interleaves the execution of these strategies, increasing the likeliness that these 'quick' proofs will be found. Secondly, previous work has established the existence of a synergistic effect when allowing proof attempts to communicate by sharing information about their inferences or clauses. The recently introduced AVATAR approach to splitting uses a SAT solver to explore the clause search space. The new architecture considers sharing this SAT solver between proof attempts, allowing them to share information about pruned areas of the search space, thus preventing them from making unnecessary inferences. Experimental results, using hard problems from the TPTP library, show that interleaving can lead to problems being solved more quickly, and that sharing the SAT solver can lead to new problems being solved by the combined strategies that were never solved individually by any existing theorem prover.

1 Introduction

This paper presents a pseudo-concurrent architecture for first-order saturation-based theorem provers. This work is a first step in a larger attempt to produce a truly concurrent architecture. This architecture allows proof attempts to cooperate and execute in a pseudo-concurrent fashion. This paper considers an instantiation of the architecture for the Vampire prover [9] but the approach is applicable to any saturation-based prover.

Modern first-order provers often use *saturation algorithms* that attempt to saturate the clausified problem with respect to a given inference system. If a contradiction is found in this clause search space then the problem is unsatisfiable. If a contradiction is not found and the search space is saturated, by a complete inference system, it is satisfiable. Even for small problems this search space can grow quickly and provers often employ heuristics to control its exploration.

A major research area is therefore the reduction of this search space using appropriate strategies. One approach is to control how the search space grows by the selection and configuration of different inferences and preprocessing steps. The aim being to find a contradiction more quickly. The notion of *redundancy* allows provers to detect parts

[⋆] Partially supported by the EPSRC grant "Reasoning for Verification and Security"

of the search space that will not be needed anymore and eliminate redundant inferences; this greatly improves the performance of provers. An additional (incomplete) heuristic for reducing the search space is Vampire's limited resource strategy [13] which discards heavy clauses that are unlikely to be processed given the remaining time and memory resources. Another useful tool in reducing search space explosion is *splitting* [8] where clauses are split so that the search space can be explored in smaller parts. A new, highly successful, approach to splitting is found in the AVATAR architecture [23], which uses a Splitting module with a SAT solver at its core to make splitting decisions.

Although most of theorem provers allow users to choose between various strategies, two issues remain. Firstly, given a problem it is generally not possible to choose a good strategy a priori even if such a strategy is implemented in the theorem prover. Secondly, whilst there are strategies that appear to be good at solving problems on average, there will be problems that can only be solved by strategies which are not good on average.

To deal with these issues, almost all modern automated theorem provers have modes that attempt multiple strategies within a given time, searching for a "fast proof". The work in this paper was motivated by the observation that whilst we are running multiple proof attempts we should allow them to cooperate. This is further supported by work in parallel theorem proving [6, 18] that discovered a synergistic effect when clauses were communicated between concurrently running proof attempts.

The pseudo-concurrent architecture presented in this paper introduces two concepts that allow proof attempts to cooperate:

1. Proof attempts are interleaved in a pseudo-concurrent fashion (Sec. 3). This allows "fast proofs" to be found more quickly.
2. The Splitting module from the AVATAR architecture is shared between proof attempts (Sec. 4). This shares information on pruned splitting branches, preventing proof attempts from exploring unnecessary branches.

The performance of automatic theorem provers can be improved in two ways. The first is to solve problems we can already solve more quickly, the second is to solve new problems that we could not solve before. The work presented here targets both ways. Results (Sec. 5) comparing this architecture to a sequential execution of Vampire are very encouraging and demonstrate that this new pseudo-concurrent architecture has great potential to make significant improvements with respect to both of our goals. It is shown that, in general, problems can be solved much faster by this new architecture. This is invaluable for applications such as program analysis or interactive theorem proving, where a large amount of proof attempts can be generated in a short time and the goal is to solve quickly as many of them as possible. Furthermore, by allowing proof attempts to communicate inconsistent splitting decisions we are able to prove new problems that were not solved before by any existing first-order theorem prover.

Further investigation is required to understand how this new architecture can be optimally utilised but it is clear that this approach can improve the general performance of saturation-based provers and solve problems beyond the reach of existing techniques.

2 Vampire and AVATAR

Vampire is a first-order superposition theorem prover. This section reviews its basic structure and components relevant to the rest of this paper. We will use the word *strategy* to refer to a set of configuration parameter values that control proof search and *proof attempt* to refer to a run of the prover using such a strategy.

2.1 Saturation algorithms.

Superposition provers such as Vampire use *saturation algorithms with redundancy elimination*. They work with a search space consisting of a set of clauses and use a collection of generating, simplifying and deleting inferences to explore this space. Generating inferences, such as superposition, extend this search space by adding new clauses obtained by applying inferences to existing clauses. Simplifying inferences, such as demodulation, replace a clause by a simpler one. Deleting inferences, such as subsumption, delete a clause, typically when it becomes redundant (see [2]). Simplifying and deleting inferences must satisfy this condition to preserve completeness.

The goal is to *saturate* the set with respect to the inference system. If the empty clause is derived then the input clauses are unsatisfiable. If no empty clause is derived and the search space is saturated then the input clauses are guaranteed to be satisfiable *only if* a complete strategy is used. A strategy is complete if it is guaranteed that all inferences between non-deleted clauses in the search space will be applied. Vampire includes many incomplete strategies as they can be very efficient at finding unsatisfiability.

All saturation algorithms implemented in Vampire belong to the family of *given clause algorithms*, which achieve completeness via a fair *clause selection* process that prevents the indefinite skipping of old clauses. These algorithms typically divide clauses into three sets, *unprocessed*, *passive* and *active*, and follow a simple *saturation loop*:

1. Add non-redundant *unprocessed* clauses to *passive*. Redundancy is checked by attempting to *forward simplify* the new clause using processed clauses.
2. Remove processed (passive and active) clauses made redundant by newly processed clauses, i.e. *backward simplify* existing clauses using these clauses.
3. Select a given clause from *passive*, move it to *active* and perform all generating inferences between the given clause and all other active clauses, adding generated clauses to *unprocessed*.

Later we will show how iterations of this saturation loop from different proof attempts can be interleaved. Vampire implements three saturation algorithms:

1. *Otter* uses both passive and active clauses for simplifications.
2. *Limited Resource Strategy (LRS)* [13] extends Otter with a heuristic that discards clauses that are unlikely to be used with the current resources, i.e. time and memory. This strategy is incomplete but also generally the most effective at proving unsatisfiability.
3. DISCOUNT uses only active clauses for simplifications.

There are also other proof strategies that fit into this loop format and can be interleaved with superposition based proof attempts. For example, instance generation [7] saturates the set of clauses with respect to the instance generation rule. As a large focus of this paper is the sharing of AVATAR, which is not compatible with instance generation, we do not consider this saturation algorithm in the following, although it is compatible with the interleaving approach (Sec. 3).

2.2 Strategies in Vampire.

Vampire includes more than 50 parameters, including experimental ones. By only varying parameters and values used by Vampire at the last CASC competition, we obtain over 500 million strategies. These parameters control

- Preprocessing steps (24 different parameters)
- The saturation algorithm and related behaviour e.g. clause selection
- Inferences used (16 different kinds with variations)

Even restricting these parameters to a single saturation algorithm and straightforward preprocessing steps, the number of possible strategies is vast. For this reason, Vampire implements a portfolio *CASC mode* [9] that categorises problems based on syntactic features and attempts a sequence of approximately 30 strategies over a five minute period. These strategies are the result of extensive benchmarking and have been shown, experimentally, to work well on unseen problems i.e. those not used for training.

2.3 AVATAR.

The search space explored by a saturation-based prover can quickly become full of long and heavy clauses, i.e those that have many literals that are themselves large. This can dramatically slow down many inferences, which depend on the size of clauses. This is exacerbated by generating inferences, which typically generate long and heavy clauses from long and heavy clauses.

To deal with this issue we can introduce *splitting*, which is based on the observation that the search space $S \cup (C_1 \vee C_2)$ is unsatisfiable if and only if both $S \cup C_1$ and $S \cup C_2$ are unsatisfiable, for variable disjoint C_1 and C_2. Different approaches to splitting have been proposed [8] and Vampire implements a new approach called AVATAR [23]. The general idea of the AVATAR approach is to allow a SAT solver to make splitting decisions. In the above case the clause $C_1 \vee C_2$ would be represented (propositionally) in the SAT solver, along with other clauses from S, and the SAT solver would decide with component to *assert* in the first-order proof search. Refutations depending on asserted components are given to the SAT solver, restricting future models and therefore the splitting search space. More details can be found elsewhere [23, 12]. Later (Section 4) we discuss how the AVATAR architecture can be used to communicate between concurrent proof attempts.

3 Interleaved Scheduling

This section introduces the concept of proof attempt interleaving and explains how it is implemented in the pseudo-concurrent architecture.

3.1 Motivation.

The intuition behind Vampire's CASC mode is that for many problems there exists a strategy that will solve that problem relatively quickly. Therefore, rather than spending five minutes using a strategy that is 'good on average' it is better to spend a few seconds on each of multiple strategies. However, strategies are attempted sequentially and a problem that can be solved quickly inside an individual strategy, may take a long time to solve within the sequence of strategies.

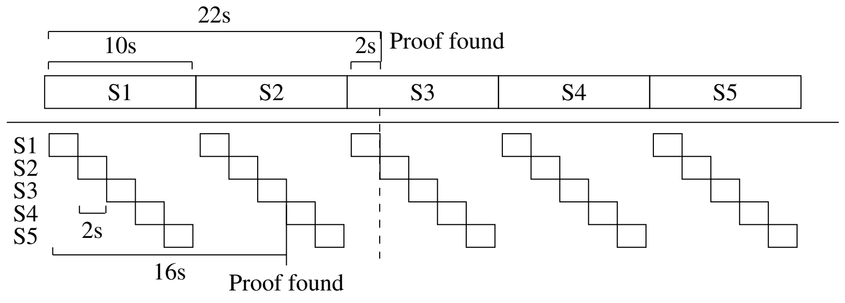

Fig. 1. An illustration of how proof attempt interleaving can prove problems faster.

By interleaving proof attempts (strategies) one can quickly reach these quick solutions. This is illustrated in Figure 3. There are five strategies and the third strategy solves the problem after 2 seconds. By interleaving the strategies in blocks of 2 seconds, the problem is solved in 16 seconds rather than 22 seconds. In reality we often have problems solved in a few deciseconds and a sequence of around 30 strategies. So the time savings have the potential to be dramatic.

3.2 Interleaving architecture.

Previously we explained how Vampire carries out proof attempts via *saturation algorithms* that iterate a *saturation loop*. The pseudo-concurrent architecture interleaves iterations of this saturation loop from different proof attempts. It is necessary that the granularity of interleaving be at this level as a proof attempt's internal data structures are only guaranteed to be consistent at the end of each iteration.

Each proof attempt is associated with a *context* that contains the structures relevant to that proof attempt. This primarily consists of the saturation algorithm with associated clause sets and indexing structures, as well as configuration parameters local to the proof attempt. In addition to proof attempt's context, there is also a *shared context*, accessible by all proof attempts, that contains a copy of the problem, the problem's signature and sort information, and a global timer. When a proof attempt is *switched in* its local context is loaded and when it is *switched out* this is unloaded.

Algorithm 1 describes the interleaving algorithm. Each strategy is loaded in from a strategy file or the command line and has an individual *local time limit*. These strategies are placed in a queue. There is a *concurrency limit* that controls the number of proof attempts that can run at any one time and a *global time limit* that restricts the run time for the whole algorithm. Global and local time limits are enforced internally by each proof attempt. Initially, the *live list* of proof attempts is populated with this number of proof attempts.

Proof attempt creation is handled by the create function. All the contexts and proof attempts are lazily initialised, so that proof attempts that will never run do not take up unnecessary memory.

The proof attempt scheduling algorithm is similar to the standard round-robin scheduling algorithm which is one of the simplest and starvation free scheduling algorithms. Scheduling is performed in circular order and without priority. Its implementation is based on a standard budgeting scheme where each proof attempt is given its own time

```
input  : A queue of strategies with local time limits
input  : A global time limit and a concurrency limit
output: Refutation, Satisfiable, Unknown or GlobalTimeLimit

live_list ← []; elapsed ← 0;
for i ← 1 to limit if size(queue) > 0 do
    proof_attempt ← create(pop(queue));
    proof_attempt.budget = 0; proof_attempt.elapsed = 0;
    add(live_list,proof_attempt);
while size(live_list) > 0 do
    time_slice ← calculate_time_slice ();
    foreach proof_attempt in live_list do
        proof_attempt.budget += time_slice;
        switchIn(proof_attempt);
        while proof_attempt.budget > 0 do
            (status, time) ← step(proof_attempt);
            elapsed += time; proof_attempt.elapsed += time;
            proof_attempt.budget = proof_attempt.budget − time;
            if elapsed > global then return GlobalTimeLimit;
            if status = Refutation or Satisfiable then return status;
            if proof_attempt.elapsed > proof_attempt.time_limit or
                status = Unknown then
                remove(live_list,proof_attempt);
                if size(queue) > 0 then
                    proof_attempt ← create(pop(queue));
                    proof_attempt.budget = 0; proof_attempt.elapsed = 0;
                    add(live_list,proof_attempt);
        switchOut(proof_attempt);
return Unknown
```

Algorithm 1: Pseudo-concurrent scheduling algorithm

budget. Ideally each proof attempt would run for a fixed time on each round. However, since it is not possible to know in advance how long a particular proof step will take, time slices for the proof attempts are dynamically calculated.

On each round a proof attempt's budget is increased by a *time slice* computed at the end of the previous round. Each proof attempt will perform multiple proof steps, decreasing its budget by the time it takes to perform each step, until its budget is exhausted. This can make a proof attempt's budget negative.

Initially, the time slice for every proof attempt is one millisecond. At the end of each round the next time slice is computed by calculate_time_slice. This calculates the average time it took each proof attempt to make a proof step and selects the smallest of the average times. Such a choice reduces number of scheduling rounds, yet providing a scheduling granularity which is fairly close to the finest. Every 1024 rounds, the time slice is doubled (thus, enlarging the granularity). Increasing the time slice reduces the scheduling overhead and lets proof attempts make reasonable progress on problems requiring a long time to solve. The constant 1024 and the time slice factor are chosen as multiples of two for better efficiency of the scheduling algorithm. Also, 1024 rounds provide approximately one second of running time for each proof attempt with the

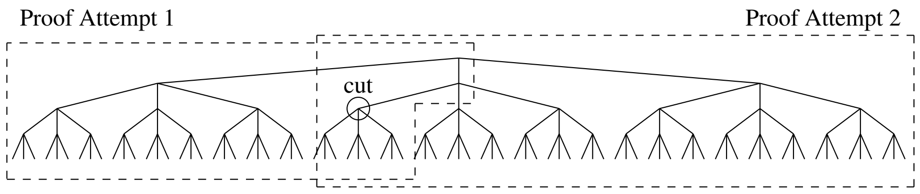

Fig. 2. Illustrating one proof attempt pruning the set of clauses explored by another.

minimal time slice of one millisecond, that seems a reasonable minimum of time for a proof attempt to make progress.

In contrast to the round-robin scheduling which gives equal time slices for execution of processes, the proof attempt scheduler only tries to give *asymptotically* equal times for running each algorithm in the execution pool. That is, all the proof attempts spend equal execution times whenever every proof attempt runs long enough.

A step of a proof attempt results in a (TPTP SZS) status. The most common returned status, which does not trigger any extra action, is *StillRunning*. If the step function reports that a proof attempt solved the problem or that all time has been used, the algorithm terminates with that status. If a proof attempt has finished running, either with a local time limit or an *Unknown* result, the proof attempt is replaced by one from the priority queue. An *Unknown* result occurs when a proof attempt is incomplete and saturates its clause set.

4 Proof Attempt Cooperation via the Splitting Module

We are interested in improving performance of a theorem prover by making proof attempts share information. A novel way of doing this, made possible by the AVATAR architecture, is to make them cooperate using the Splitting module. This is done in the following way: clauses generated by different proof attempts are passed to the same Splitting module, and thus processed using the same SAT solver, storing propositional clauses created by all proof attempts.

This idea is based on two observations:

1. SAT solving is cheap compared with first-order theorem proving. Therefore, there should be no noticeable degradation of performance as the SAT solver is dealing with a much larger set of clauses.
2. A SAT solver collecting clauses coming from different proof attempts can make a proof attempt exploit fewer branches.

4.1 Motivation.

The clause search space of a problem can be viewed as a (conceptual) splitting tree where each clause is a node and each branch represents splitting that clause into components. The leaves of the splitting tree are the *splitting branches* introduced earlier. Recall that a problem is unsatisfiable if a contradiction is found in each of these splitting branches.

In our pseudo-concurrent architecture the different proof attempts are exploring the same (conceptual) splitting tree. A proof attempt's clause space (the processed clauses,

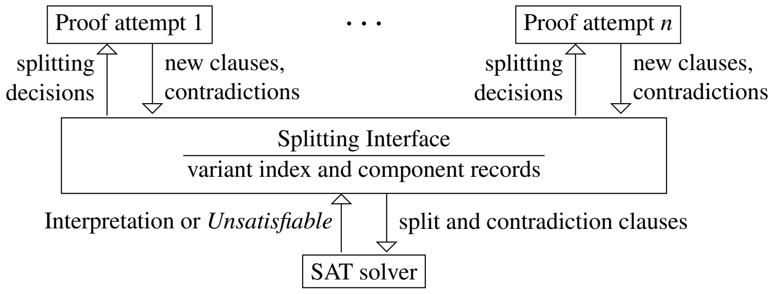
Proof attempt 1
. . .
Proof attempt n
splitting decisions
new clauses, contradictions
splitting decisions
new clauses, contradictions
Splitting Interface
variant index and component records
Interpretation or Unsatisfiable
split and contradiction clauses
SAT solver

Fig. 3. Multiple proof attempts utilising the same Splitting module

see Section 2) covers part of this tree and is determined by the inferences and heuristics it employs. Some proof attempts will have overlapping clause spaces.

By sharing the SAT solver, proof attempts are able to share splitting branches that contain contradictions. Figure 2 illustrates this effect. Proof attempts 1 and 2 have overlapping clause spaces. If one of the proof attempts show that the circled node (representing a clause component) cannot be true then the other proof attempt does not need to consider this branch. Even better, inconsistency of a branch can be established by clauses contributed by more than one proof attempt.

Cooperating in this way reduces the number of splitting branches each proof attempt needs to explore. However, it can also change the behaviour of a strategy as splitting branches may be explored in a different order. Although all splitting branches must be explored to establish unsatisfiability, the order of exploration can impact how quickly a contradiction is found in each splitting branch.

4.2 Organising cooperation.

In reality, splitting requires more structures than simply the SAT solver, as described below. We refer to all structures required to perform splitting as the Splitting module. In this section we describe how the proof attempts interact with the Splitting module. The details of how this is done for a single proof attempt are described elsewhere [23]; we focus on details relevant to multiple proof attempts.

Each proof attempt interacts with the Splitting module independently via a Splitting interface. It passes to the interface clauses to split and splitting branch contradictions it finds. It receives back splitting decisions in the form of clause components to assert (and those to retract). Cooperation is organised so that the assertions it is given by the Splitting module relate only to that proof attempt's clause space.

As illustrated in Figure 3, the Splitting interface sits between the proof attempts and the SAT solver that makes the splitting decisions. Its role is twofold. Firstly, it must consistently transform clause components into propositional variables to pass to the SAT solver. Secondly, it must transform the interpretation returned by the SAT solver into splitting decisions to pass to the proof attempts. To achieve these two roles, the Splitting interface keeps a variant index and component records.

The variant index ensures that components that are equivalent up to variable renaming and symmetry of equality are translated into the same propositional variable. Importantly, this variant index is common across all proof attempts, meaning that a

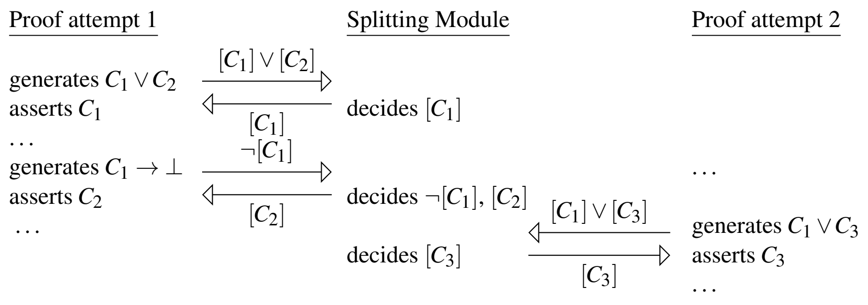

Fig. 4. Illustrating cooperation via the Splitting Module

clause C_1 generated in one proof attempt that is a variant of clause C_2 generated in a different proof attempt will be represented by the same propositional variable in the SAT solver. A (non-trivial) index is used because checking clause equivalence up to variable renaming is a problem equivalent to graph isomorphism [13].

Component records store important information about the status of the component. Ordinarily they track information necessary to implement backtracking, i.e. which clauses have been derived from this component and which clauses have been reduced assuming this component. This is important, but not interesting for this work. In our pseudo-concurrent architecture we also track which proof attempt has passed a clause containing this component to the Splitting interface. This allows the Splitting interface to restrict splitting decisions sent to a proof attempt to components generated by that proof attempt.

It is possible to allow more information to flow from the Splitting module to the proof attempts. It would be sound to assert the whole splitting branch in each proof attempt. However, this might pollute the clause space, preventing them from making suitable progress. An approach we have not yet explored is to employ heuristics to select components that should be 'leaked' back to proof attempts.

4.3 Example of cooperation.

Figure 4 illustrates how cooperation can avoid repetition of work. In this scenario the first proof attempt generates a clause consisting of two components C_1 and C_2 and the Splitting module first decides to assert C_1. After some work the first proof attempt shows that C_1 leads to a contradiction and passes this information to the Splitting module. The Splitting module decides that C_2 must hold since C_1 cannot hold. Later, a second proof attempt generates a clause consisting of the components C_1 and C_3. These are passed to the Splitting module and, as C_1 has been contradicted, C_3 is asserted immediately. Here the second proof attempt does not need to spend any effort exploring any branch containing C_1. The component C_1 could have been a large clause and the repetition of the potentially expensive work required to refute it is avoided.

5 Evaluation

We evaluate our pseudo-concurrent architecture by comparing it to the sequential version of Vampire. We are interested in how it currently performs, but also in behaviours that would suggests future avenues for exploration for this new and highly experimental architecture.

5.1 Experimental setup.

We select 1747 problems[1] from the TPTP library [19]. These consist of Unsatisfiable or Theorem problems of 0.8 rating or higher containing non-unit clauses. The rating [20] indicates the hardness of a problem and is the percentage of (eligible) provers that cannot solve a problem. For example, a rating of 0.8 means that only 20% of (eligible) provers can solve the problem. Therefore, we have selected very hard problems. Note that the rating evaluation does not consider every mode of each prover, so it is possible that a prover can solve a problem of rating 1 using a mode not used in this evaluation. Therefore, we include problems of rating 1 that we know are solvable by some prover, i.e. Vampire. Out of all problems used, 358 are of rating 1.

In this paper we focus on a single set of 30 strategies drawn from the current CASC portfolio mode, however we have explored different sets of strategies and the results presented here are representative. All strategies employ default preprocessing and use a combination of different saturation algorithms, splitting configurations and inferences. We consider a 10 second run of each strategy.

We compare two approaches that use these 30 strategies:

1. The *concurrent* approach utilises the pseudo-concurrent architecture giving a 10 second local time limit to each proof attempt.
2. The *sequential* approach executes the strategies in a predetermined *random* order without any sharing of clauses or resources, i.e. they are run in different provers.

Experiments were run on the StarExec cluster [17], using 160 nodes. The nodes used contain a Intel Xeon 2.4GHz processor. The default memory limit of 3GB was used for sequential runs and this was appropriately scaled for concurrent runs, i.e. for 30 strategies this was set to 90GB.

5.2 Results.

Figure 5 plots the number of problems solved by the *concurrent* and *sequential* approaches against the time taken to solve them.

This demonstrates that the *concurrent* approach solved more problems than the *sequential* approach for running times not exceeding 290 seconds, reaching the maximal difference of 125 problems at 20 seconds. The lead of the *concurrent* approach is more substantial from the start to about the 85th second. From 207 to 290 seconds both approaches were roughly on par, but, after 290 seconds, the *sequential* approach started to gain their lead. So the whole running time can be divided into four intervals: from 0 to 85 seconds, between 85 and 207 seconds, from 207 to 290 seconds, and after 290 seconds.

The order of strategies used in the *sequential* approach does not effect these results in general. Experiments (not reported here) showed that other random orderings led to a similar curve for the *sequential* approach. However, the reverse ordering does allow the *sequential* approach to take the lead at first as one strategy placed (randomly) at the end of the sequence can solve many problems. This reflects the obvious observation that a single strong strategy can outperform many concurrent strategies at first but the

[1] A list of the selected problems, the executable of our prover, and the results of the experiments are available from http://vprover.org

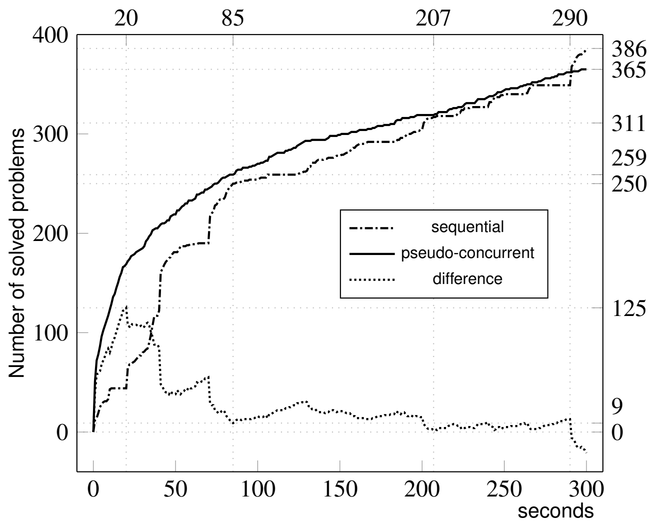

Fig. 5. Number of problems solved by a sequence of proof attempts and the pseudo-concurrent architecture with respect to time.

results show that after a short time the combined effect of many strategies leads to more problems being solved.

The *sequential* approach solved 11 problems of rating 1 and the *concurrent* approach solved 14, where 6 were not solved by the *sequential* approach. Of these problems, the *concurrent* approach solved 5 that had not previously been solved by Vampire using any mode. During other experiments, not reported here, the concurrent approach also solved 5 problems not previously solved by any prover, including Vampire. This is significant as Vampire has been run extensively for several years on the TPTP library with many different strategies, while we made very few experiments with the concurrent version.

Overall, the *concurrent* approach solved 63 problems unsolved by the *sequential* approach, which solved 84 unsolved by the *concurrent* approach. Of the problems solved by both approaches, the *concurrent* approach solved problems 1.53 times faster than the *sequential* approach on average. This represents clear advantages for applications where solving problems fast is of key importance. As scheduling is not effected by the global time limit, Figure 5 can be used to give the number of problems solved for any time limit less than 300s.

Below we discuss how these results can be interpreted in terms of interleaving and sharing the Splitting module.

5.3 The impact of interleaving.

We contribute much of the success of the *concurrent* approach on the first interval shown in Figure 5 to the advantage of interleaved proof attempts discussed in Section 3.

One can compute the ideal interleaving performance by inspecting the shortest time taken to solve a problem by the sequential proof attempts. The *concurrent* approach roughly follows this ideal in the first interval. For example, after 85 seconds the *con-*

current approach has solved 259 problems and after 2.83 seconds (approximately how long each strategy should have run for) the sequential proof attempts had solved 273 problems. However, this deviation suggests that interleaving is not perfect.

After 85 seconds, the positive effects of interleaving seem to diminish. These effects could be attributed to sharing the Splitting module. However, a separate experiment, shown in Figure 6, illustrates that there is inherent overhead in the interleaving approach. In this experiment 10 of the strategies were forced not to use splitting and we compare sequential and pseudo-concurrent execution. The interleaving effects are positive to begin with. But we would expect the two approaches to solve the same number of problems overall, and any difference can only be explained by overhead introduced by interleaving, as splitting is not used.

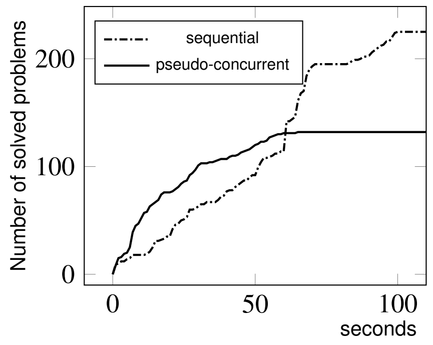

Fig. 6. Comparing pseudo-concurrent and sequential architectures with no splitting.

The cost of switching proof attempts is very small. However, changing proof attempts can have a large impact on memory locality. The average memory consumption for the *concurrent* approach is just under 4GB, compared with the 80MB (0.08GB) used by each sequential strategy on average. Large data structures frequently used by a proof attempt (such as indices) will not be in cache when it is switched in, leading to many cache faults. This is aggravated by the frequent context switching.

This slowdown due to memory overhead can explain the loss of 84 problems. Of these problems, 76% are solved by the *sequential* approach after 2.85 seconds (the end of the first interval) and 23% after 6.9 seconds (the end of the second interval). The implementation of the new architecture did not focus on minimising memory consumption and future work targeting this issue could alleviate these negative effects.

5.4 The impact of sharing AVATAR.

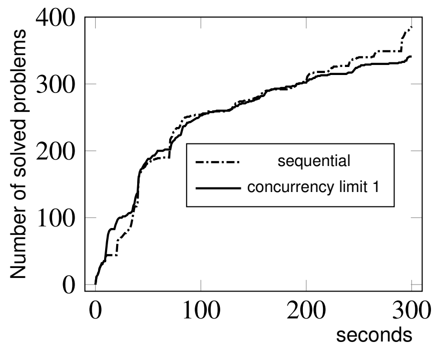

Fig. 7. Running with concurrency limit 1 shows AVATAR sharing effects.

We have seen the positive impact strategy interleaving can have, but also the negative impact that running multiple proof attempts in the same memory space can have. Strategy interleaving cannot account for all of the positive results. There were 63 problems solved by the *concurrent* approach and not by the *sequential* approach. This makes up 17% of problems solved and can be attributed to sharing the Splitting module as non-cooperating interleaved proof-attempts are expected to solve the same problems as the proof attempts running sequentially.

Figure 7 shows the results of running the *concurrent* approach with a concurrency limit of 1 (i.e. running one strategy at a time). This is very similar to the *sequential* approach. However, the *concurrent* approach solves new problems early on. After 20 seconds it has solved 49 problems that the *sequential* approach has not solved. This is only possible though the additional information shared via the Splitting module.

Furthermore, individual strategies solve many new problems. In one case, a strategy that solves no problems in the *sequential* approach solves 45 problems within the *concurrent* approach with concurrency limit one. This can be explained by the Splitting module letting this proof attempt skip many unnecessary splitting branches to quickly reach a contradiction.

This behaviour is also seen when there is no concurrency limit. One strategy solves 16 problems when sharing the Splitting module, compared with the sequential case; on average local strategies solve 4.8 problems that their sequential counterparts do not solve. This demonstrates that sharing the Splitting module improves the progress of individual strategies generally.

Note that sharing the Splitting module can introduce additional overhead as each call to the SAT solver will be dealing with more propositional variables. This is confirmed by timing results. On average, sequential proof attempts spent roughly 23% of total running time in the SAT solver whilst proof attempts within the pseudo-concurrent architecture spent roughly 28%. The represents an increase of approximately 20%.

5.5 Summary.

The experiments in this section show the following:

– Interleaving strategies can find the *quick* proofs, but multiple proof attempts sharing the memory space limits this effect. Further work allowing proof attempts to share certain data structures should alleviate this to some extent.
– Sharing the Splitting module can lead to many new problems being solved that were not solved without this sharing, including very hard problems.

6 Related Work

There are two areas of work related to this new architecture. We first consider the usage of multiple strategies sequentially and then discuss links with parallel theorem proving.

Sequential usage of strategies. All modern theorem provers that are competitive in the CASC competition utilise some form of strategy scheduling. The idea was first implemented in the Gandalf prover [22] which dynamically selects a strategy depending on results of performed inferences. Other solvers have similar approaches to Vampire's CASC mode, for example, in the E prover [14] the selection of strategies and generation of schedules for a class of problems is also based on the previous performance of the prover on similar problems. The idea of strategy scheduling has been developed further in the E-MaLeS 1.1 scheduler [10] where schedules for the E prover are produced using kernel-based learning algorithms.

Parallel theorem proving. The approach taken in this work has many aspects in common with the area of parallel theorem proving (see [4, 21, 24] for extensive overviews on problems, architectures, and approaches) including our motivation and methods.

Firstly, previous work with distributed provers (e.g. PARTHEO [15]) and parallel provers (e.g. DISCOUNT [6]) has shown that overall better performance and stability can be achieved by running multiple proof attempts in parallel. Running several theorem provers in parallel without any communication between them also shown to exhibit such a behaviour [3]. As many strategies are explored in parallel, concurrent proof attempts can solve and guarantee answers to more problems. Additionally, it was shown that parallel provers also behave more predictably when run repeatedly on same set of problems. These effects are similar to the consequences of interleaving proof attempts that is described in the paper.

As mentioned in the beginning of this paper, previous work in parallel theorem proving has also observed a synergistic effect [18] for proof attempts that communicate by passing information about their inferences. In this previous work a parallel execution of proof attempts was able to solve a number of hard problems which could not be solved by any individual proof attempt. This effect was also observed in the DISCOUNT parallel theorem prover [6], based on a generic framework for concurrent theorem proving called the *teamwork approach* [5]. Here clause communication is handled by a complex management scheme that passes 'outstanding' clauses between proof attempts. We have observed a similar effect with our sharing of the Splitting module.

Our approach, and the previous parallel theorem provers, run multiple proof attempts concurrently. An alternative approach is to parallelise the internal workings of the prover. In 1990, Slaney and Lusk [16] proposed a parallel architecture for closure algorithms and investigated its application to first-order theorem proving. The work considered how a generalisation of the internal saturation loop in theorem provers can be turned into a multi-threaded algorithm. The resulting ROO theorem prover [11] provides a *fine-grained* splitting of the closure algorithms into concurrent tasks.

7 Conclusion

The presented results show a promising direction in developing concurrent provers. As a future goal we consider developing a truly concurrent thread-safe architecture for saturation-based first-order theorem provers. The main challenge in this respect is to make such a concurrent prover efficient. The advantages of scheduled interleaving of proof attempts demonstrated by the experimental results prove that this is possible. Co-operation of proof attempts via common interfaces like AVATAR is also shown to be fruitful: it was shown that the pseudo-concurrent architecture could utilise this sharing to prove previously unsolvable problems. Also, we have demonstrated that problems can be solved much faster, making the architecture attractive for applications, such as program analysis and interactive theorem proving. Although developing a truly concurrent and efficient prover is an ultimate goal, there are a few other steps can be done to improve the current pseudo-concurrent architecture. One is to develop prover configuration methods which take into account interleaved execution of the strategies. This could utilise machine learning techniques. Another is to allow scheduling modulo priorities assigned to the proof attempts and maximise efficiency of the prover by pre-configuring such parameters. A third step, which is more important and difficult, is to expand co-operation scheme of the proof attempts allowing more useful information to be shared between them with minimal overhead. These directions are interesting, require many experiments, and, hopefully, will lead to new architectures of theorem provers ensuring better stability and efficiency.

References

1. *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*, vol. 449 of *LNCS*. Springer, 1990.
2. L. Bachmair and H. Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, vol. I, chapter 2, pp. 19–99. Elsevier Science, 2001.
3. S. Böhme and T. Nipkow. Sledgehammer: Judgement Day. In *IJCAR 2010*, vol. 6173 of *LNCS*, pp. 107–121. Springer, 2010.
4. M. Bonacina. A taxonomy of parallel strategies for deduction. *Ann. Math. Artif. Intell.*, 29(1-4):223–257, 2000.
5. J. Denzinger and M. Kronenburg. Planning for distributed theorem proving: The teamwork approach. In *KI-96*, vol. 1137 of *LNCS*, pp. 43–56. 1996.
6. J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT — a distributed and learning equational prover. *J. Autom. Reasoning*, 18(2):189–198, 1997.
7. H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *Proc. LICS'03*, pp. 55–64, 2003.
8. K. Hoder and A. Voronkov. The 481 ways to split a clause and deal with propositional variables. In *Proc. CADE-24*, vol. 7898 of *LNCS*, pp. 450–464. 2013.
9. L. Kovács and A. Voronkov. First-order theorem proving and vampire. In *CAV 2013*, vol. 8044 of *LNCS*, pp. 1–35, 2013.
10. D. Kühlwein, S. Schulz, and J. Urban. E-MaLeS 1.1. In *CADE-24*, vol. 7898 of *LNCS*, pp. 407–413, 2013.
11. E. Lusk and W. McCune. Experiments with ROO: A parallel automated deduction system. In *Dagstuhl Seminar on Parallelization in Inference Systems*, vol. 590 of *LNCS*, pp. 139–162. Springer, 1990.
12. G. Reger, M. Suda, and A. Voronkov. Playing with AVATAR. In *Proc. CADE'15*. 2015.
13. A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comp.*, 36(1-2):101–115, 2003.
14. S. Schulz. System Description: E 1.8. In *Proc. LPAR-19*, vol. 8312 of *LNCS*, 2013.
15. J. Schumann and R. Letz. PARTHEO: A high-performance parallel theorem prover. In *Proc. CADE* [1], pp. 40–56.
16. J. K. Slaney and E. L. Lusk. Parallelizing the closure computation in automated deduction. In *Proc. CADE* [1], pp. 28–39.
17. StarExec, https://www.starexec.org.
18. G. Sutcliffe. The design and implementation of a compositional competition-cooperation parallel ATP system. In *Proc. IWIL-2*, number MPI-I-2001-2-006 in MPI für Informatik, Research Report, pp. 92–102, 2001.
19. G. Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.
20. G. Sutcliffe and C. Suttner. Evaluating General Purpose Automated Theorem Proving Systems. *Artificial Intelligence*, 131(1-2):39–54, 2001.
21. C. B. Suttner and J. Schumann. Chapter 9 — Parallel automated theorem proving. In *Parallel Processing for Artificial Intelligence*, vol. 14 of *Machine Intelligence and Pattern Recognition*, pp. 209 – 257. North-Holland, 1994.
22. T. Tammet. Gandalf. *J. Autom. Reasoning*, 18(2):199–204, 1997.
23. A. Voronkov. AVATAR: The architecture for first-order theorem provers. In *Proc. CAV'14*, vol. 8559 of *LNCS*, pp. 696–710. 2014.
24. A. Wolf and M. Fuchs. Cooperative parallel automated theorem proving. Technical Report SFB Bereicht 342/21/97, Technische Universität München, 1997.