piCoq: Parallel Regression Proving for Large-Scale Verification Projects

Karl Palmskog
University of Texas at Austin, USA
palmskog@acm.org

Ahmet Celik
University of Texas at Austin, USA
ahmetcelik@utexas.edu

Milos Gligoric
University of Texas at Austin, USA
gligoric@utexas.edu

ABSTRACT

Large-scale verification projects using proof assistants typically contain many proofs that must be checked at each new project revision. While proof checking can sometimes be parallelized at the coarse-grained file level to save time, recent changes in some proof assistant in the LCF family, such as Coq, enable fine-grained parallelism at the level of proofs. However, these parallel techniques are not currently integrated with regression proof selection, a technique that checks only the subset of proofs affected by a change. We present techniques that blend the power of parallel proof checking and selection to speed up regression proving in verification projects, suitable for use both on users' own machines and in workflows involving continuous integration services. We implemented the techniques in a tool, piCoq, which supports Coq projects. piCoq can track dependencies between files, definitions, and lemmas and perform parallel checking of only those files or proofs affected by changes between two project revisions. We applied piCoq to perform regression proving over many revisions of several large open source projects and measured the proof checking time. While gains from using proof-level parallelism and file selection can be considerable, our results indicate that proof-level parallelism and proof selection is consistently much faster than both sequential checking from scratch and sequential checking with proof selection. In particular, 4-way parallelization is up to 28.6 times faster than the former, and up to 2.8 times faster than the latter.

CCS CONCEPTS

• **Theory of computation** → *Logic and verification*; • **Software and its engineering** → *Software evolution*;

KEYWORDS

Proof assistants, verification, parallelism, proof engineering, Coq

ACM Reference Format:
Karl Palmskog, Ahmet Celik, and Milos Gligoric. 2018. piCoq: Parallel Regression Proving for Large-Scale Verification Projects. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3213846.3213877

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213877

1 INTRODUCTION

Many large-scale verification projects rely on *proof assistants* to construct and check formal proofs [25]. Representative projects target critical software domains, e.g., compilers [36], operating systems [32], file systems [16], and distributed systems [37, 61], or formalize mathematical theories [10, 28].

Results certified by proof assistants are highly trustworthy, but establishing properties demands significant time investment by sophisticated users to guide the proof effort. When such projects are modified, previously proven properties must be *reestablished*, since even small changes can break a critical step in a proof—this process of *regression proving* may take anywhere from seconds to tens of minutes [13], and in extreme cases, several days [30].

Analogously to when building and testing software engineering projects, productivity suffers in verification projects when regression proving takes an inordinate amount of time. One important technique for speeding up software builds and tests on commodity multi-core hardware is *parallelization* [8, 12, 33]. Due to recent changes in popular proof assistants in the LCF family, such as Coq [5] and Isabelle/HOL [57], parallelization of proof checking is now possible not only at the coarse-grained level of *files* (via build systems such as `make`) but also at the fine-grained level of individual *proofs*. As we have argued in previous work [13], a formal proof of some program property can be viewed as representing many (possibly an infinite number of) tests. Based on this analogy, coarse-grained parallel proof checking intuitively corresponds to test suite parallelization at the *test class* level in Java-like languages, while fine-grained parallel proof checking corresponds to parallelization at the *test method* level.

Coarse-grained parallel proof checking is used in many Coq verification projects, e.g., when building such projects via the OCaml Package Manager (OPAM) [19, 41]. One important reason for widespread use of coarse-grained parallelism is that proof checking using proof assistants in the LCF family is deterministic and occurs in a predictable runtime environment. Additionally, when surveying 260 publicly available Coq projects on GitHub, each with more than 10k lines of code (LOC), we found that ∼20% of these projects can also leverage fine-grained parallel proof checking, two years after support was added to Coq.

Regression proof selection [13], a technique that avoids checking proofs unaffected by changes as a project evolves, is orthogonal to parallel proof checking. Unfortunately, at present, regression proof selection is not integrated with parallel proof checking. This means that large-scale projects must generally perform regression proving from scratch, in particular when using (as they often do) continuous integration services (CISs) such as Travis CI [54]. Even with parallelism using many cores, the resulting long proof checking time can be a burden to users.

In this paper, we describe techniques that blend proof checking parallelization and selection to speed up regression proving in large-scale verification projects. As we demonstrate in our evaluation, these novel techniques are superior to *legacy* (state-of-the-art) techniques even on users' own machines, but are particularly effective when used in CISs. We believe that our techniques can alleviate the cost to productivity and trust in evolving large-scale verification projects caused by long proof checking times [1], and release untapped potential in multi-core hardware for regression proving.

This paper makes the following contributions:

- ⋆ **Techniques**: We propose novel techniques that integrate parallel proof checking and selection to speed up regression proving in evolving verification projects using proof assistants. Along one axis, we consider coarse-grained and fine-grained proof checking parallelism. Along the other axis, we consider selection at both the file and proof levels, i.e., we check only those files or proofs that are affected by changes. The result is a *taxonomy* of regression proving techniques that also includes legacy techniques.
- ⋆ **Tool**: We implemented our techniques in a tool, dubbed PICOQ, which supports Coq projects. PICOQ relies on a collection of extensions to the Coq proof checking toolchain, several of which originate in the ICOQ tool [14]. We provide a version of PICOQ on the following URL: http://cozy.ece.utexas.edu/icoq.
- ⋆ **Evaluation**: We performed an empirical study to measure the effectiveness of our regression proving techniques using PICOQ. We used many revisions of several large-scale open source Coq projects, and measured the proof checking times for our techniques and legacy techniques. Our results show that speedups can be substantial from adopting proof-level parallelism and file selection, but that improvements vary significantly across projects, and may even be absent. However, combined proof-level parallelism and proof selection is consistently much faster than both sequential checking from scratch (legacy) and sequential checking with proof selection (ICOQ). Specifically, in a CIS environment, we obtain a speedup of up to 28.6× with 4-way fine-grained parallelism and proof selection compared to the former, while the speedup compared to the latter is up to 2.8×.

2 COQ BACKGROUND

The Coq proof assistant consists of, on the one hand, an implementation of a small and powerful purely functional programming language, and on the other hand, a system for specifying properties about programs and proving them. Coq is based on a type theory called the Calculus of Inductive Constructions [44], where both programs and propositions about programs are *types* inhabited by *terms*. In effect, this property puts program development and proving on the same footing for Coq users.

In a typical workflow for a Coq-based project, users interactively construct tentative proof terms for propositions using operations called *tactics*. Propositions are only accepted as proven after Coq's type checker has been run successfully on a proof term. Absent inconsistent axioms and frontend issues, the user need only trust that the comparatively small type checking kernel is correctly implemented and compiled to trust that proven propositions really hold. Figure 1 illustrates the interactive proof development process.

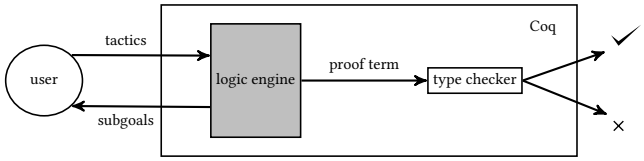

Figure 1: Coq interactive proof development overview.

2.1 Coq Proof Checking Toolchain

Definitions of functions and propositions processed by Coq are written in the Gallina language, and reside in files ending in `.v`. The Coq batch proof checking (compilation) tool, `coqc`, takes a `.v` file as input and, by default, produces a `.vo` file as output that contains full binary representations of processed Gallina constructs, including proofs. Since files may depend on other files, checking all proofs in a Coq project requires some form of dependency analysis. The standard `coq_makefile` tool generates a Makefile which, by default, calls the `coqdep` tool for this purpose [53]. `coqdep` builds a dependency graph for all input `.v` files based on simple syntactic analysis of `Require` commands in files (similar to `import` statements in Java), which indicate direct dependency at the file level. When proof checking is performed via the generated Makefile, the dependency graph is used to process `.v` files with `coqc` in some allowed order. The Makefile also enables timestamp-based incremental regression proving in a Coq project, as well as spawning of parallel proof-checking processes. Note that such proof-checking parallelism is fundamentally restricted by the file dependency graph; for example, if this graph is a path (has no branches) there will be no parallel checking at all.

2.2 Asynchronous Proof Checking in Coq

Coq version 8.5, the first stable release to include architectural changes to support a *document-oriented* interaction model [5], introduced the option to *quick-compile* `.v` files to the binary `.vio` format, a process which avoids checking (and emitting representations of) proofs that are indicated as *opaque* by ending with `Qed`. Figure 2 illustrates the new `.vio` proof checking workflow made possible by Coq's document-oriented model. Only the type (proposition) of an opaque lemma, i.e., not the body proof term, can be referenced in other parts of a Coq development, whence type checking of all such terms can normally be performed in complete isolation, and thus in parallel. Specifically, `.vio` files contain *proof-checking tasks*, which can be performed individually by issuing a `coqc` command referencing the task identifier. A Coq user can depend on more rapidly produced `.vio` files in lieu of `.vo` files in most developments, but must then assume that all proofs are correct.

Asynchronous proof checking has two important applications in large-scale Coq projects. First, it enables *regression proof selection*, i.e., the possibility of checking only affected proofs after each new project revision [13, 22]. Second, it enables *fine-grained* parallel proof checking that can make better use of commodity multi-core hardware than file-level parallel checking [5, 57]. Specifically, `coqc` includes the option `-schedule-vio-checking`, which takes as arguments (i) an upper bound on the number of parallel processes, and (ii) a list of `.vio` files whose proof tasks to check in parallel. However, note that there is no way to specify *subsets* of proof tasks in files to check in parallel. In contrast to purely Makefile-based task

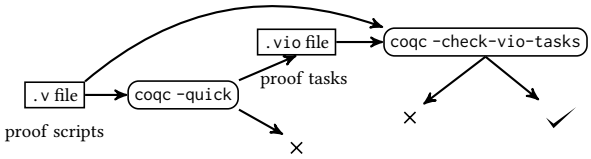

Figure 2: Coq asynchronous proof checking workflow.

parallelism, which often fails to utilize the requested number of parallel processes throughout proof checking due to file dependency restrictions, the degree of parallelism for fine-grained checking depends directly on how checking of individual proofs is scheduled on the parallel operating system processes.

Coq uses a notion of *sections* to organize common assumptions made in a collection of lemmas, say, that equality on a type `A` is decidable (`A_eq_dec`). A lemma may reference one or more such assumptions, which then become quantified variables that must be instantiated when the lemma is referenced outside of the section. However, by default, Coq only determines the used section variables of a lemma when the end of the section is reached. This means that the final type (assertion) of the section lemma is not known when considered in isolation, whence its proof cannot be immediately checked as an asynchronous task. To get around this problem, Coq allows section lemmas to be annotated with the assumptions they use (e.g., `Proof using A_eq_dec`). The required annotations can be derived from metadata produced by Coq during compilation of source files to `.vo` files [52], and then inserted back into the source files. In the evaluation of our techniques (Section 6), we used this approach to automatically add annotations to all revisions of the projects under study.

2.3 ICoq and Regression Proof Selection

ICoq is a tool for (sequential) regression proof selection in Coq [13, 14]. ICoq tracks dependencies among both files and proofs in order to check only those proofs affected by changes to a project, potentially saving significant time in comparison to checking everything from scratch. ICoq processes Coq projects in three phases (each similar to the corresponding phase in regression test selection tools [26, 43, 47]): *analysis*, *proof checking*, and *dependency collection*. In the analysis phase, ICoq detects files and proofs that are affected by changes made since the last run of ICoq. In the proof checking phase, ICoq uses Coq's toolchain for asynchronous processing to check the proofs selected in the analysis phase (but not other proofs). Finally, in the collection phase, ICoq obtains the new dependencies that will be used in the next run of the analysis.

3 RUNNING EXAMPLE

We use the small Coq library of list functions and lemmas shown in Figure 3 to illustrate our techniques; code is extracted from the StructTact project [51]. The Coq standard library contains a function `remove` that, when given a decision procedure for equality for a type `A`, removes a single element from a list of that type. The file `ListUtil.v` contains two lemmas about the `remove` function. `Dedup.v` defines a function `dedup` that omits any duplicates from the argument list, and a lemma about this function. `RemoveAll.v` defines a function `remove_all` that removes *all* elements identical to the given element from a list, and two lemmas about this function.

```
Require Import List. Import ListNotations.

Lemma remove_preserve : ∀ (A : Type) A_eq_dec (x y : A) xs,
  x ≠ y → In y xs → In y (remove A_eq_dec x xs).
Proof.
induction xs; simpl; intros.
- intuition.
- case A_eq_dec; intros.
  * apply IHxs; subst; intuition.
  * intuition; subst; left; auto.
Qed.

Lemma in_remove : ∀ (A : Type) A_eq_dec (x y : A) xs,
  In y (remove A_eq_dec x xs) → In y xs.
Proof.
induction xs; simpl; intros; auto.
destruct A_eq_dec; simpl in *; intuition.
Qed.
```

ListUtil.v

```
Require Import List ListUtil. Import ListNotations.

Fixpoint dedup (A : Type) A_eq_dec (xs : list A) : list A :=
match xs with
| [] ⇒ []
| x :: xs ⇒
  if in_dec A_eq_dec x xs then dedup A A_eq_dec xs
  else x :: dedup A A_eq_dec xs
end.

Lemma remove_dedup : ∀ A A_eq_dec (x : A) xs,
  remove A_eq_dec x (dedup A A_eq_dec xs) =
  dedup A A_eq_dec (remove A_eq_dec x xs).
Proof.
induction xs; intros; auto; simpl.
repeat (try case in_dec; try case A_eq_dec; simpl; intuition);
  auto using f_equal.
- exfalso. apply n0. apply remove_preserve; auto.
- exfalso. apply n. apply in_remove in i; intuition.
Qed.
```

Dedup.v

```
Require Import List ListUtil. Import ListNotations.

Fixpoint remove_all A A_eq_dec (to_delete l : list A) : list A :=
match to_delete with
| [] ⇒ l
| d :: ds ⇒ remove_all A A_eq_dec ds (remove A_eq_dec d l)
end.

Lemma remove_all_in : ∀ A A_eq_dec ds l x,
  In x (remove_all A A_eq_dec ds l) → In x l.
Proof.
induction ds; simpl; intros; intuition.
eauto using in_remove.
Qed.

Lemma remove_all_preserve : ∀ A A_eq_dec ds l x,
  ¬ In x ds → In x l → In x (remove_all A A_eq_dec ds l).
Proof.
induction ds; simpl; intros; intuition auto using remove_preserve.
Qed.
```

RemoveAll.v

Figure 3: Example Coq project.

As indicated by the `Require` commands and by direct references inside proof scripts in `Dedup.v` and `RemoveAll.v`, the proofs of lemmas in these files depend on lemmas in `ListUtil.v`. For example, the proof of the lemma `remove_all_in` in `RemoveAll.v` depends on the lemma `in_remove` in `ListUtil.v`. Figure 4 shows both the file-level and proof-level dependencies of the project. File dependencies are illustrated by solid line arrows, and dependencies among definitions, lemmas, and proofs by dashed arrows. As

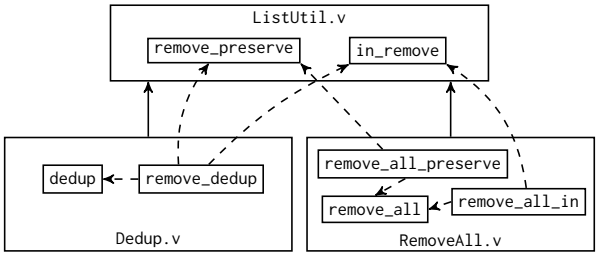

Figure 4: Dependencies in the Coq project in Figure 3.

```
Lemma in_remove : ∀ (A : Type) A_eq_dec (x y : A) xs,
  In x (remove A_eq_dec y xs) → In x xs.
Proof.
induction ys; simpl; intros; auto.
destruct A_eq_dec; simpl in *; intuition.
Qed.
```

Figure 5: Revised version of lemma `in_remove` in the file `ListUtil.v` in Figure 3, with changed line highlighted.

```
Fixpoint dedup (A : Type) A_eq_dec (xs : list A) : list A :=
match xs with
| [] ⇒ []
| x :: xs ⇒
  let tail := dedup A A_eq_dec xs in
  if in_dec A_eq_dec x xs then tail else x :: tail
end.
```

Figure 6: Revised version of function `dedup` in the file `Dedup.v` in Figure 3, with changed lines highlighted.

indicated in the figure, the proofs of lemmas in both `Dedup.v` and `RemoveAll.v` depend on the utility lemmas in `ListUtil.v`, but not all of the former depend on all of the latter. Nevertheless, the default Coq proof-checking toolchain checks all proofs and writes `.vo` files whenever a change is made to some utility lemma in `ListUtil.v`.

In contrast, the asynchronous proof-checking toolchain, complemented by a tool such as ɪCoq, allows avoiding many instances of proof checking when making changes to `ListUtil.v`. For example, suppose the maintainers of the project change the definition of the lemma `in_remove` to the one in Figure 5 where variables `x` and `y` are swapped (highlighted). To ensure that all previously proven properties in the library hold, the proofs of `remove_dedup` and `remove_all_in` both need to be checked in addition to `in_remove`. To perform these tasks, we first compile the `.v` files to `.vio` files (which elides checking of all lemmas ending in `Qed`). Then, we issue individual proof-checking commands separately for the three lemmas, which reveals that their proofs can be reestablished.

As another example, consider the proposed change (highlighted) to the `dedup` function in Figure 6. Intuitively, the change only removes some code duplication by using a `let` expression to encapsulate the recursive call, i.e., the meaning of the function is preserved. Since this change only affects `Dedup.v`, this is the only file that we need to recompile to a `.vo` file when relying on the default toolchain (assuming a recent `ListUtil.vo` file is available). With the asynchronous proof-checking toolchain, we need to recompile `Dedup.v` to a `.vio` file, and then issue a proof checking command for the lemma `remove_dedup`.

Table 1: Modes for Regression Proving in Coq.

Parallelization	Selection		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	n/a
Proof level	`p·none`	`p·file`	`p·icoq`

Table 2: `f·none` Mode for Coq Project in Figure 3; Same-Phase Tasks Can Run in Parallel.

Phase	Task	Definitions and Lemmas
1	`ListUtil.vo`	`remove_preserve`, `in_remove`
2	`Dedup.vo`	`dedup`, `remove_dedup`
2	`RemoveAll.vo`	`remove_all`, `remove_all_in`, `remove_all_preserve`

4 TECHNIQUES

This section describes our taxonomy of regression proving techniques, which includes both legacy techniques and our proposed novel techniques. To concretize the presentation, we describe the techniques as *proof checking modes* for Coq, as defined in Table 1.

A fundamental choice when checking Coq proofs is whether to use default compilation or quick-compilation of source files. Opting for default compilation means that all proof-checking must be performed top-down according to the file-level dependency graph, which also restricts the (file-level) parallelism according to this graph. With default compilation, a user can either perform *no selection* (`f·none`), i.e., check the whole project from scratch, or coarse-grained *file-level selection* (`f·file`), where only proofs in files affected by actual changes are checked. With quick-compilation, the previous two forms of selection (`p·none` and `p·file`) are complemented by fine-grained *proof-level selection*, where only individual proofs affected by changes are checked (`p·icoq`).

We consider two execution environments for each proof checking mode: CI-Env and LO-Env. CI-Env describes an environment that uses a Continuous Integration Service (CIS) [31], e.g., Travis CI, to check proofs. Note that a CIS checks proofs in a clean environment for each revision. LO-Env describes an environment where developers use their local machines to check proofs. Note that file timestamps and generated files, not present in version control, are preserved in LO-Env, but not in CI-Env.

We next describe the details of each mode and discuss variants of each mode for CI-Env and LO-Env.

`f·none`: This legacy mode embodies the approach used in the default Coq proof-checking toolchain with `coq_makefile`. Since all files are fully checked for every revision, there is no difference between running this mode in LO-Env and CI-Env. Many large-scale projects on GitHub use this mode in their Travis CI jobs, e.g., Verdi [61]; we therefore used it as the CIS baseline when investigating the speedup from sequential proof selection using ɪCoq [13].

On one hand, this mode has no overhead from proof checking task management and tracking dependencies across revisions. On the other hand, parallelism is restricted by the file dependency graph, and there is overhead from writing (possibly large) `.vo` files to disk. Table 2 illustrates how parallel checking can be performed using the `f·none` mode for the example project in Figure 3.

Table 3: p·none Mode for Coq Project in Figure 3; Same-Phase Tasks Can Run in Parallel and Proof Tasks (to be Run in a Later Phase) are in Bold.

Phase	Task	Definitions and Lemmas
1	ListUtil.vio	**remove_preserve**, **in_remove**
2	Dedup.vio	dedup, **remove_dedup**
2	RemoveAll.vio	remove_all, **remove_all_in**, **remove_all_preserve**
3	checking	remove_preserve
3	checking	in_remove
3	checking	remove_dedup
3	checking	remove_all_in
3	checking	remove_all_preserve

p·none: This legacy mode embodies the approach used in the asynchronous proof-checking toolchain introduced by Barras et al. [5]. As in f·none, which is the closest comparable alternative, there is no difference between running the mode in LO-Env and CI-Env. Although coq_makefile generates tasks to use this mode, we found that ~20% of 260 projects on GitHub that we analyzed (with more than 10k LOC each) can use p·none properly without modification. The reason for this percentage not being higher is the requirement to annotate proofs inside sections, as explained in Section 2.

On one hand, proof checking in p·none is not restricted by the file dependency graph, and there is no overhead from writing proof terms to disk or tracking dependencies across revisions. On the other hand, this mode requires .vio file compilation of (annotated) .v files according to the file dependency graph, and has overhead from managing individual proof checking tasks. Table 3 illustrates the maximum possible parallelism of this mode for the project in Figure 3. In the table, lemmas whose proofs are exempt from checking in a task are marked in bold. Note that the possible degree of parallelism is greater than for f·none due to isolated checking of proofs. The degree of parallelism can be adjusted downwards by moving proofs from one checking task to another.

f·file: This novel mode adds file-level selection to f·none by only compiling *affected* .v files to .vo files between revisions, with directly modified files determined by comparing current file checksums to previous checksums. As such, f·file suffers from the overhead of maintaining a file dependency graph using coqdep, but not from managing proof-checking tasks. In LO-Env, this mode corresponds to the baseline we compared sequential proof selection against in previous work using ICoq [13], which modeled developers incrementally checking projects on their local machines. In CI-Env, dependency graph and file checksum metadata must be explicitly persisted. Moreover, additional .vo files (those on which modified files depend) may have to be compiled in CI-Env compared to LO-Env for the same change to a project, since previously compiled .vo files are not available. Same as for f·none, the degree of parallelism is restricted by project file dependencies, and all proof terms in selected files are written to disk.

Suppose we are working with the project in Figure 3 and perform the change indicated in Figure 5. In f·file for both LO-Env and CI-Env, regression proving would then entail (re)compiling all .v files to .vo files, with parallelism as in Table 2. If we instead perform the change in Figure 6, f·file in LO-Env only recompiles Dedup.v into Dedup.vo, without any possibility of parallelism. In CI-Env, both ListUtil.v and Dedup.v are compiled into .vo files (sequentially).

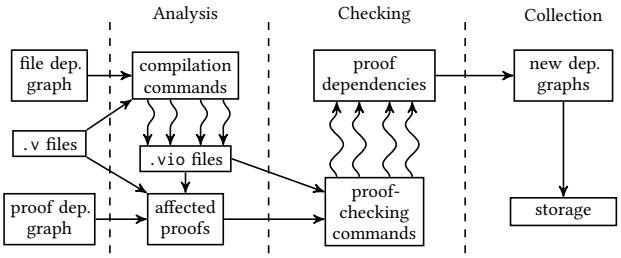

Figure 7: p·icoq workflow/phases with 4-way parallelism.

Table 4: p·icoq Mode for Change in Figure 5 to Project in Figure 3; Same-Phase Tasks Can Run in Parallel, and Proof Tasks (to be Run in a Later Phase) are in Bold.

Phase	Task	Definitions and Lemmas
1	ListUtil.vio	**remove_preserve**, **in_remove**
2	Dedup.vio	dedup, **remove_dedup**
2	RemoveAll.vio	remove_all, **remove_all_in**, **remove_all_preserve**
3	checking	in_remove
3	checking	remove_dedup
3	checking	remove_all_in

p·file: This novel mode adds file-level selection to p·none, using the same analysis of file changes and file dependencies as in f·file. Consequently, p·file has overhead both from maintaining a file dependency graph and for management of proof-checking tasks, but is not restricted by file dependency graph for proof-checking parallelization. After determining and quick-compiling the necessary files (using file checksums), the mode uses the coqc command -schedule-vio-checking described in Section 2.2. Running this command will typically check many unaffected proofs needlessly. However, p·file does not write proof terms to disk.

Suppose we are working with the project in Figure 3 and perform the change indicated in Figure 5. In p·file for both LO-Env and CI-Env, regression proving would then entail recompiling all .v files to .vio files and then reestablishing all lemmas in all .vio files, via the same parallelism as in Table 3. If we instead perform the change in Figure 6, p·file in LO-Env entails recompiling Dedup.v into Dedup.vio, and then checking all proof tasks in that file, i.e., checking remove_dedup asynchronously. p·file in CI-Env needs to compile both ListUtil.v and Dedup.v into .vio files, but runs the same coqc command to check remove_dedup.

p·icoq: This novel mode, which is the most sophisticated one in our taxonomy, combines fine-grained parallelism with proof selection, and corresponds to ICoq [13] when used sequentially (i.e., with one proof-checking process). The mode incurs overhead both from proof task management and from tracking of proof-level dependencies. However, it features proof checking parallelism unrestricted by file dependencies and can elide checking unaffected proofs regardless of their location in a file. Moreover, no proof terms are written to disk. In both LO-Env and CI-Env, p·icoq relies on file checksumming to first locate changed files, which are then subject to more detailed impact analysis at the level of proofs. Figure 7 illustrates the workflow for p·icoq in the case of four parallel jobs for quick-compilation and fine-grained proof checking.

Table 5: Projects Used in the Evaluation.

Project	URL	SHA	LOC	#Revs.	#Files	#Proof Tasks
Coquelicot	https://scm.gforge.inria.fr/anonscm/git/coquelicot/coquelicot	680ca587	38260	24	29	1660
Finmap	https://github.com/math-comp/finmap	baec7ba0	5661	23	4	959
Flocq	https://scm.gforge.inria.fr/anonscm/git/flocq/flocq	4161c990	24786	23	40	943
Fomegac	https://github.com/skeuchel/fomegac	7a654d7c	2637	14	13	156
Surface Effects	https://github.com/esmifro/surfaceeffects	3450e4b7	9621	24	15	289
Verdi	https://github.com/uwplse/verdi	15be6f61	56147	24	222	2756
$\sum$	n/a	n/a	137112	132	323	6763
Avg.	n/a	n/a	22852.00	22.00	53.83	1127.16

Suppose we are working with the project in Figure 3 and perform the change indicated in Figure 5. In `p·icoq` for both LO-Env and CI-Env, regression proving would first entail recompiling all `.v` files to `.vio` files, and then proving asynchronously `in_remove`, `remove_dedup`, and `remove_all_in` (skipping the unaffected `remove_preserve` and `remove_all_preserve`). Parallelism for this case is illustrated in Table 4. If we instead perform the change in Figure 6, `p·icoq` in LO-Env entails recompiling `Dedup.v` into `Dedup.vio`, and then checking `remove_dedup` asynchronously. In contrast, CI-Env regression proving requires quick-compiling both `ListUtil.v` and `Dedup.v`, and then checking `remove_dedup`.

5 IMPLEMENTATION

We implemented the modes described in Section 4 in a tool dubbed PICOQ, written in OCaml, Java, and bash. Since Coq developments are not upwards or downwards compatible in general, we target Coq version 8.5 to support the largest range of project revision histories susceptible to asynchronous proof checking and fine-grained parallelism; we expect no fundamental issues with supporting the latest (8.8) and future Coq versions.

We extended `coqc` (the official Coq compiler) with the new option `-schedule-vio-task-depends-checking`. As first argument, it takes an upper bound on the number of parallel processes, and then a list of proof task definitions (pairs of `.vio` file names and task identifiers). `coqc` with `-schedule-vio-task-depends-checking` performs parallel proof checking of all indicated tasks in the same way as the official `-schedule-vio-checking` option (which only takes whole files as arguments), but also outputs the dependencies of each processed proof individually. This dependency data can then be used by PICOQ to select affected proofs in the next revision.

We rely on the same Coq plugins and extensions as ICOQ [13, 14], adapted for parallel checking. We also use the graph-based analysis from ICOQ to find affected files and proofs across revisions, which is similar to the approach used in build systems such as Bazel [6] and CloudMake [21]. We use Java's task executor facilities [27] for parallel compilation of `.vo` and `.vio` files via `coqc` commands.

6 EVALUATION

To assess the efficacy of our proposed techniques on large, evolving verification projects, we answer the following research questions:

RQ1: How effective, in terms of proof checking time, is coarse-grained parallel regression proving (file-level parallelism) without any selection, i.e., `f·none`?

RQ2: How effective, in terms of proof checking time, is fine-grained parallel regression proving (proof-level parallelism) without any selection, i.e., `p·none`?

RQ3: How effective, in terms of proof checking time, is coarse-grained parallel regression proving with selection at the level of files using PICOQ, in CISs and on developers' own machines, i.e., `f·file` in CI-Env and LO-Env?

RQ4: How effective, in terms of proof checking time, is fine-grained parallel regression proving with selection at the level of files using PICOQ, in CISs and on developers' own machines, i.e., `p·file` in CI-Env and LO-Env?

RQ5: How effective, in terms of proof checking time, is fine-grained parallel regression proving with selection at both the level of files and individual proofs using PICOQ, in CISs and on developers' own machines, i.e., `p·icoq` in CI-Env in LO-Env?

We run all experiments on a 4-core Intel Core i7-6700 CPU @ 3.40GHz machine with 16GB of RAM, running Ubuntu 17.04. We limit the number of parallel processes to be at or below the number of physical cores; this avoids the problem of drawing clear conclusions about speedups when using virtual cores (hyper-threading).

6.1 Verification Projects Under Study

Table 5 shows the list of Coq projects used in our study; all projects are publicly available. We selected projects based on (a) public availability of their revision history during principal development, (b) compatibility of their revision history with Coq version 8.5, (c) their size and popularity, and (d) tractability of their build process; the latter was necessary for a successful experimental setup. For each project, we show the name, the repository URL, the last revision/SHA we used for our experiments, and the number of lines of Coq code (LOC) for the last revision, as reported by `cloc` [18]. Note that since Coq projects have different development paces and added support for Coq 8.5 at different points in time, our revision ranges are not all from the same time period.

Coquelicot: Coquelicot is a library for real number analysis [10], containing results about limits, derivatives, integrals, etc.

Finmap: Finmap is a library of definitions and results about finite sets and finite maps [17], based on the Mathematical Components library [20, 29].

Flocq: Flocq is a library that formalizes floating-point arithmetic in several representations [11], e.g., as described in the IEEE-754 standard. Flocq is used in the CompCert verified C compiler to reason about programs which use floating-point operations [9].

Require: $\mathcal{P}$ a project under study
Require: κ the number of revisions
Require: ε a development environment
Require: η range of number of parallel jobs
1: **procedure** ExperimentProcedure($\mathcal{P}, \kappa, \eta, \varepsilon$)
2: Clone($\mathcal{P}$.url)
3: **for all** $\iota \in \{1, \ldots, \eta\}$ **do**
4: **for all** $\rho \in$ LatestRevisions($\kappa, \mathcal{P}$) **do**
5: Checkout(ρ)
6: Configure($\mathcal{P}, \varepsilon, \iota$)
7: SelectAndCheck($\mathcal{P}$)
8: **end for**
9: **end for**
10: **end procedure**

Figure 8: Experiment procedure.

Fomegac: This project contains a formalization of a version of the formal system F_ω and the corresponding metatheory, such as type safety results [23].
Surface Effects: This project formalizes a functional programming language of "surface effects" with operations on mutable state [46], including its operational semantics and metatheory (typability, effect soundness and correctness).
Verdi and Verdi Raft: Verdi is a framework for verification of implementations of distributed systems [60]. While the framework is not currently tied to any one particular verification project, it was initially bundled with a verified implementation of the Raft distributed consensus protocol [61]. Each revision comprises over 50k LOC, making Verdi one of the largest publicly available software verification projects.

6.2 Variables

We manipulate three independent variables in our experiments: *proof checking mode*, *development environment*, and (maximum) *number of parallel jobs*. The proof checking modes and environments are as described in Section 4. The number of parallel jobs ranges from 1 to 4. As a dependent variable we consider only the proof checking time.

6.3 Experiment Procedure

Figure 8 shows our experiment procedure for collecting the data necessary to answer our research questions. The inputs to the procedure include one of the projects used in the study, number of revisions to use in the experiment, a development environment, and a range of number of parallel jobs. In the initial step (line 2), the procedure clones the project repository from the URL in Table 5. Next, the procedure iterates over the range of parallel jobs, starting from one, until the upper bound η is reached. For a particular number of parallel jobs, the procedure iterates over κ revisions, from the oldest to the newest revision. In each iteration of the inner loop, the procedure (a) obtains a copy of the project for the current revision (line 5), (b) configures the project (in preparation for proof checking), and (c) selects proofs or files that are affected by changes and checks them. While executing the procedure, we log the time for each step of the procedure and the number of executed proofs; we report the former in this paper and use the latter to check correctness of our experiments.

It is important to observe that some piCoq modes require persisting dependency metadata files between each revision. One way to do this in a CIS is to use built-in caching facilities [49]. Since the dependency data is small, we do not associate any overhead with persisting these files, even in CI-Env.

6.4 Results

We illustrate some of the data collected with our procedure in Figure 9. The plots (for Coquelicot and Fomegac) show, for every revision of the projects, the proof checking time in all modes when using 4 cores; we do not show similar plots for other projects due to space limitations.

Table 6 lists the total proof checking times for all projects and modes. The first column shows the name of the project, and the second column shows the name of the mode. Columns three to six show the results for each number of parallel jobs. Recall that parallel checking does not necessarily use all available parallel processes all the time, since tasks may depend on other tasks.

RQ1: For Verdi in `f•none`, going from sequential to 4-way parallel coarse-grained checking jobs brings a speedup of 3.3×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.5×, 1.2×, 1.7×, 1.1×, and 1.0×, respectively. These latter modest improvements may be due to restrictions in project file dependency graphs; yet, having two parallel jobs nearly always gives a sizable speedup compared to sequential checking.

RQ2: For Verdi in `p•none`, going from sequential to 4-way parallel fine-grained checking brings a speedup of 3.2×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 2.0×, 1.8×, 2.0×, 1.5×, and 2.2×, respectively, indicating greater potential for improvements per core than `f•none`. Speedups compared to 4-way parallelism via `f•none` are noteworthy for some projects, e.g., Fomegac (1.4×) and Surface Effects (2.2×). However, Finmap and Flocq are consistently slower to check with `p•none` than `f•none`; this may be due to both projects having many short-running proofs.

RQ3: For Verdi in `f•file-CI`, going from sequential to 4-way parallel coarse-grained checking brings a speedup of 3.0×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.5×, 1.0×, 1.7×, 1.0×, and 1.0×, respectively. The corresponding speedups in LO-Env are essentially the same for most projects. Compared to `f•none` with 4-way parallelization, the speedup for Verdi in CI-Env is 4.3×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.1×, 2.0×, 2.3×, 1.2×, and 1.0×, respectively. This indicates that `f•file` can be an improvement over `f•none` in CISs.

RQ4: For Verdi in `p•file-CI`, going from sequential to 4-way parallel coarse-grained checking brings a speedup of 3.2×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.8×, 2.3×, 1.9×, 1.5×, and 2.0×, respectively. LO-Env gives essentially similar speedups. Compared to `p•none` with 4-way parallelization, the speedup for Verdi in CI-Env is 4.7×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.5×, 3.7×, 2.5×, 1.3×, and 1.0×, respectively.

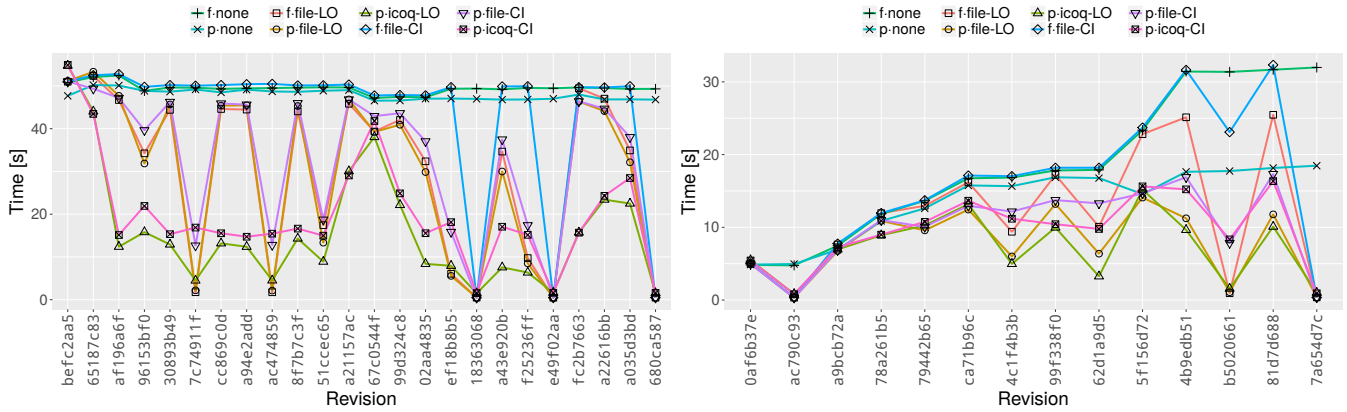

Figure 9: Comparison of proof checking times for different modes across revisions of Coquelicot (left) and Fomegac (right). The plots show the proof checking time using four parallel jobs.

RQ5: For Verdi in `p·icoq-CI`, going from sequential (ICoq) to 4-way parallelism gives a speedup of 2.8×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.5×, 1.6×, 1.6×, 1.4×, and 1.8×, respectively. Compared to other modes, this mode gives the lowest proof checking times for nearly all projects, both in LO-Env and CI-Env. Compared to `p·file-CI`, which is arguably the most reasonable comparison, `p·icoq-CI` with 4-way parallelization gives speedups for Verdi, Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects of 1.8×, 1.6×, 1.3×, 1.2×, 1.1×, and 1.6×, respectively. The speedup for Verdi in `p·icoq-CI` with 4-way parallelization compared to sequential `f·none` is notable: 28.6×.

7 DISCUSSION

Regression proof mode choices: As shown by our results, different modes have widely varying effects on the proof-checking time for different projects. For example, most projects see small or no improvement when switching from `f·none` to `p·none`, while Verdi shows improvement even for only one job. We believe the efficacy of fine-grained parallel checking is directly related to the frequency of long-running proofs in a project, which is relatively high in Verdi, as indicated, e.g., by the number of proof script lines (around 30k). Long-running proofs may allow for fine-grained parallelism to overcome its inherent overhead from task management.

Another consideration when adopting PICoq for a Coq project is trust in the toolchain. Even though a developer may trust the Coq proof checker, regression proving tools like PICoq add several additional layers on top of this checker to trust. Yet, `f·file` and `p·file`, which essentially only rely on file checksumming and `coqdep` for extracting information from `.v` files, require less trust than `p·icoq`, which performs complex analysis of Coq files via plugins. On the other hand, users may opt for quicker feedback using `p·icoq` while still running `f·file` or `p·file` later for higher assurance.

When choosing between `f·file` and `p·file`, a key factor is the file dependency graph of the project. A low number of files itself implies restrictions, but even with many files, dependencies may force certain restricted orders for compilation of `.vo` files; this is particularly apparent for Surface Effects, but also for Coquelicot. Such projects may opt for `p·file` despite its higher overhead. The choice of mode may also depend on the number of available cores in CI-Env. As shown by the `f·none` results for Coquelicot, there may be a breaking point where no further number of jobs and cores decrease coarse-grained proof checking time (but two jobs/cores generally give a large improvement up from just one). In contrast, `p·none` (and therefore `p·file`) can continue to improve with additional cores where `f·none` cannot, but with diminishing marginal returns. The unusual organization of Verdi definitions and proofs into many different files based on type classes and their instances appears to be highly conducive to parallel compilation of files, in a way similar to how module abstraction (functorization) facilitates separate compilation in functional languages [2, 35].

Support for generic definitions: Sozeau and Tabareau introduced support for *generic* Coq definitions that can be used across universes of types [48]. However, Coq's toolchain for asynchronous proof processing ignores universe constraints, since such constraints must be checked for consistency at the global level [52]. This precludes safe use of the modes using fine-grained parallel proof-checking (Section 4) in projects that make heavy use of universe polymorphism. One way to address this problem is to leverage the `coqc` option `-vio2vo`, which produces `.vo` files from argument-passed `.vio` files, by asynchronously building the proofs for all proof tasks (possibly in parallel). Once the `.vo` files are produced, they can be loaded into Coq to evaluate universe consistency globally [52]. This approach enables fine-grained parallel proof checking with file selection, but proof selection remains out of reach.

8 THREATS TO VALIDITY

External: Our results may not generalize to all Coq projects. To mitigate this threat, we selected projects that vary in size, number of files, number of proofs, and proof checking time.

Our experiments are executed on a single hardware platform and may not be reproducible. To mitigate this threat, we ran a subset of experiments on our local machines. Although the absolute numbers are not the same as those reported in Section 6, proof checking time ratios among our developed techniques remained the same.

We used up to 24 revisions per project. The results may differ for different windows of revisions or longer histories. We selected the

Table 6: Total Execution Time in Seconds for All Modes and Projects for Different Number of Jobs.

Project	Mode	Time [jobs=1]	Time [jobs=2]	Time [jobs=3]	Time [jobs=4]
Coquelicot	f·none	1807.49	1226.33	1186.49	1187.29
	p·none	2319.39	1400.08	1260.69	1150.87
	f·file-LO	1043.01	745.41	724.32	725.55
	p·file-LO	1221.63	814.89	747.11	708.50
	p·icoq-LO	552.05	407.84	396.55	384.43
	f·file-CI	1587.52	1086.38	1049.53	1051.73
	p·file-CI	1405.10	885.67	802.13	786.88
	p·icoq-CI	732.60	528.78	505.49	479.79
Finmap	f·none	646.77	549.91	549.50	549.62
	p·none	1377.30	800.71	766.60	758.22
	f·file-LO	236.77	232.74	232.64	232.08
	p·file-LO	459.30	334.14	225.59	196.15
	p·icoq-LO	227.31	179.55	153.42	148.11
	f·file-CI	279.04	274.72	274.09	274.29
	p·file-CI	473.59	346.19	234.34	205.13
	p·icoq-CI	258.72	191.73	164.81	159.54
Flocq	f·none	892.54	525.44	516.65	512.46
	p·none	1173.08	702.60	614.19	579.45
	f·file-LO	319.22	197.82	190.47	189.68
	p·file-LO	390.66	244.64	227.00	216.38
	p·icoq-LO	274.63	194.55	183.71	175.62
	f·file-CI	370.22	230.27	223.21	221.15
	p·file-CI	430.79	260.61	233.30	231.80
	p·icoq-CI	320.37	216.64	206.25	196.08
Fomegac	f·none	278.31	261.13	261.37	261.66
	p·none	293.33	222.66	199.18	191.88
	f·file-LO	171.80	165.41	165.32	165.44
	p·file-LO	176.15	135.29	117.64	109.17
	p·icoq-LO	153.69	121.88	109.95	101.33
	f·file-CI	228.11	220.48	220.31	220.50
	p·file-CI	210.82	168.78	151.20	142.48
	p·icoq-CI	188.18	155.84	142.80	134.75
Surface Effects	f·none	8712.55	8633.68	8627.13	8629.28
	p·none	8465.24	4863.77	4067.83	3902.42
	f·file-LO	7820.60	7796.43	7781.69	7787.54
	p·file-LO	7599.11	4408.28	3432.82	3237.19
	p·icoq-LO	3669.35	2288.69	2033.12	2014.79
	f·file-CI	8714.85	8652.68	8651.51	8656.17
	p·file-CI	7695.15	4595.89	4047.15	3769.91
	p·icoq-CI	4116.04	2723.55	2404.59	2351.96
Verdi	f·none	35713.55	19157.52	13449.78	10947.00
	p·none	33032.34	17675.38	12692.18	10275.42
	f·file-LO	7405.04	4009.03	3007.21	2473.82
	p·file-LO	6917.62	3633.05	2564.76	2059.44
	p·icoq-LO	3339.82	1866.04	1370.73	1160.28
	f·file-CI	7577.00	4127.27	3059.70	2536.91
	p·file-CI	7040.39	3781.12	2704.12	2203.67
	p·icoq-CI	3542.08	1990.56	1478.36	1247.65

latest revisions of each project that could be built with Coq version 8.5 (the version supported by the implementation of ICoq, which PICoq extends). Although several projects have long histories, we faced issues similar to those frequently faced when building projects written in other languages [55], e.g., code that cannot be compiled.

Internal: Our implementation of PICoq, as well as our evaluation infrastructure, may contain bugs. To mitigate this, we tested it on changes to small example projects where the expected regression proving outcome was easy to check manually.

The mode for checking that relies on proof selection (`p·icoq`) is subject to the same limitations as ICoq, e.g., with respect to tactic language dependencies and parameterized modules (functors). To trust the results of `p·icoq`, a user must trust Coq's asynchronous proof-checking toolchain as well as the ICoq technique and its implementation in terms of Coq plugins, extensions to `coqc`, and the ICoq proof-level dependency graph analysis.

Construct: We implemented parallel regression proving with selection only for a single proof assistant. Although our techniques and taxonomy should be applicable to other proof assistants (e.g., Isabelle/HOL [40]), future work should confirm the applicability.

9 RELATED WORK

Candido et al. [12] investigated the prevalence of test suite parallelization in open-source Java projects, and found that uncertainty about outcomes (e.g., due to flaky tests [7, 8, 38, 39]) sometimes prevented adoption. In contrast to a unit test, whether a proposed proof passes or fails checking is deterministic. Gambi et al. [24] introduced a tool, named CUT, for automatic cloud-based unit testing. They also introduced a tool named O!Snap for generating test plans to optimize cloud-based test execution. Chakraborty and Shah [15] proposed an approach to derive a test execution plan, based on available resources and test dependencies. Unlike work on distributing test execution, PICoq combines proof selection and parallelization on a single machine. Distributing proof checking is an interesting future direction.

Regression proof selection was inspired by regression test selection (RTS) techniques [43, 47, 62]. Although initial work on RTS was mostly focused on fine-grained dependencies (e.g., statements, basic blocks, and methods) [47, 50, 63], recent work has shown that coarse-grained dependencies (e.g., files and classes) may provide more savings in terms of end-to-end execution time (due to a lightweight program analysis) [26, 34, 42]. Interestingly, and opposite to findings for RTS, we find that fine-grained proof selection and parallelization provides the most savings for proof checking; combining coarse-grained and fine-grained selection for proof checking is a future direction.

Coq's 1970s precursor LCF was based on synchronous, sequential interaction between a user and the proof tool [58]. This legacy is reflected in Coq's read-eval-print loop, and by extension, in the top-down interaction with Coq files in classic interfaces such as Proof General [3]. Architectural changes in Isabelle towards a document-oriented asynchronous interaction model were pioneered by Wenzel [58]. Efforts to bring asynchronous interaction to Coq were initiated by Wenzel [56] and Barras et al. [4], resulting in a new Isabelle-inspired document-oriented interaction model and support for asynchronous proof processing in Coq 8.5 [5].

Support for parallelism in construction and checking of proofs to exploit multi-core hardware has been studied previously in several proof assistants, notably Isabelle [57] and ACL2 [45]. Isabelle leverages the support for threads in its "host" compiler, Poly/ML, to spawn proof checking tasks processed by parallel workers. Using a notion of *proof promises*, proofs that require previous unfinished result can proceed normally and become finalized when extant tasks terminate. Isabelle also includes a build system with integrated support for checking of proofs and management of parallel workers. ACL2 uses the support for thread-based parallelism in LISP systems to, e.g., perform parallel proof discovery and fine-grained proof case checking. The lack of native threads in Coq's host language, OCaml, prevents similar low-cost fine-grained parallelism [57].

Our approach to proof selection builds closely on previous work on ICoq [13, 14] which, however, did not consider any form of parallelism and only selection of individual proofs—not of files. Wenzel [59] describes how to scale Isabelle for batch processing of long-running large proofs using both parallelism and other features.

In many Coq projects, the default build process is via a Makefile that sometimes supports file-level parallelism. When projects are packaged via OPAM [19, 41], this parallelism is typically exposed and is leveraged whenever a user installs the package. PICoq offers similar parallelism both in LO-Env and CI-Env, and in contrast to OPAM, supports incremental checking of projects. In addition, PICoq exposes fine-grained parallel proof checking.

Barras et al. [5] investigated the speedup of fine-grained parallel proof checking compared to coarse-grained file-based proof checking for a single Coq project when building one revision from scratch. PICoq combines their approach with several forms of selection in regression proving, and we evaluate speedups for a wide range of projects with many revisions.

10 CONCLUSION

We presented a taxonomy comprising both state-of-the-art and novel techniques for parallel, incremental regression proving in large-scale verification projects, and their implementation for the Coq proof assistant in the tool PICoq. In particular, PICoq is suitable for use in continuous integration systems running on multi-core hardware to quickly verify a project or find failing proofs. By tracking dependencies, PICoq can avoid checking unaffected files or proofs as changes are made, and check the remaining affected proofs in parallel.

Our evaluation shows that switching from sequential checking in a CIS environment (commonly using Travis CI) to 4-way proof-level parallelization and proof selection with PICoq can lead to speedups of up to 28.6×. Compared to previous work on sequential proof checking with proof selection in a CIS environment with ICoq, 4-way proof-level parallelization using PICoq yields speedups of up to 2.8×. These results indicate the potential of our techniques and PICoq to increase the productivity of proof engineers.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments and Tevfik Bultan for shepherding this paper; Pengyu Nie, Chenguang Zhu, and Zachary Tatlock for their feedback on this work. This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1566363 and CCF-1652517.

REFERENCES

[1] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *Trans. Program. Lang. Syst.* 37, 2 (2015), 7:1–7:31.

[2] Andrew W. Appel and David B. MacQueen. 1994. Separate Compilation for Standard ML. In *Conference on Programming Language Design and Implementation*. 13–23.

[3] David Aspinall. 2000. Proof General: A Generic Tool for Proof Development. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 38–43.

[4] Bruno Barras, Lourdes del Carmen González Huesca, Hugo Herbelin, Yann Régis-Gianas, Enrico Tassi, Makarius Wenzel, and Burkhart Wolff. 2013. Pervasive Parallelism in Highly-Trustable Interactive Theorem Proving Systems. In *Intelligent Computer Mathematics*. 359–363.

[5] Bruno Barras, Carst Tankink, and Enrico Tassi. 2015. Asynchronous Processing of Coq Documents: From the Kernel up to the User Interface. In *International Conference on Interactive Theorem Proving*. 51–66.

[6] BazelBlogWebPage 2018. Bazel - Blog. (2018). https://bazel.io/blog/.

[7] Jonathan Bell and Gail E. Kaiser. 2014. Unit test virtualization with VMVM. In *International Conference on Software Engineering*. 550–561.

[8] Jonathan Bell, Gail E. Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient Dependency Detection for Safe Java Test Acceleration. In *International Symposium on Foundations of Software Engineering*. 770–781.

[9] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. 2013. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *Symposium on Computer Arithmetic*. 107–115.

[10] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. 2015. Coquelicot: A User-Friendly Library of Real Analysis for Coq. *Mathematics in Computer Science* 9, 1 (2015), 41–62.

[11] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *Symposium on Computer Arithmetic*. 243–252.

[12] Jeanderson Candido, Luis Melo, and Marcelo d'Amorim. 2017. Test Suite Parallelization in Open-Source Projects: A Study on Its Usage and Impact. In *Automated Software Engineering*. 838–848.

[13] Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2017. iCoq: Regression Proof Selection for Large-Scale Verification Projects. In *Automated Software Engineering*. 171–182.

[14] Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2018. A Regression Proof Selection Tool For Coq. In *International Conference on Software Engineering, Demo*. 117–120.

[15] Soham Sundar Chakraborty and Vipul Shah. 2011. Towards an Approach and Framework for Test-execution Plan Derivation. In *Automated Software Engineering*. 488–491.

[16] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a High-performance Crash-safe File System Using a Tree Specification. In *Symposium on Operating Systems Principles*. 270–286.

[17] Cyril Cohen. 2017. finmap. (2017). https://github.com/math-comp/finmap.

[18] Al Danial. 2017. cloc - counts blank lines, comment lines, and physical lines of source code in many programming languages. (2017). https://github.com/AlDanial/cloc.

[19] Coq development team. 2018. Coq Package Index. (2018). https://coq.inria.fr/opam/www/.

[20] MathComp development team. 2018. Mathematical Components Project. (2018). https://math-comp.github.io/math-comp/

[21] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's Distributed and Caching Build Service. In *International Conference on Software Engineering, Software Engineering in Practice*. 11–20.

[22] Alexander Faithfull, Jesper Bengtson, Enrico Tassi, and Carst Tankink. 2016. Coqoon. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 316–331.

[23] FomegacGit 2016. Fomegac Git repository. (2016). https://github.com/skeuchel/fomegac.git.

[24] Alessio Gambi, Sebastian Kappler, Johannes Lampel, and Andreas Zeller. 2017. CUT: Automatic Unit Testing in the Cloud. In *International Symposium on Software Testing and Analysis*. 364–367.

[25] Herman Geuvers. 2009. Proof assistants: History, ideas and future. *Sadhana* 34, 1 (2009), 3–25.

[26] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *International Symposium on Software Testing and Analysis*. 211–222.

[27] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice*. Addison-Wesley.

[28] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. 2013. A Machine-Checked Proof of the Odd Order Theorem. In *International Conference on Interactive Theorem Proving*. 163–179.

[29] Georges Gonthier and Assia Mahboubi. 2010. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning* 3, 2 (2010), 95–152.

[30] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, and Roland Zumkeller. 2017. A Formal Proof of the Kepler Conjecture. *Forum of Mathematics, Pi* 5 (2017).

[31] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Automated Software Engineering*. 426–437.

[32] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Symposium on Operating Systems Principles*. 207–220.

[33] Colin J. W. Kushneryk and Paul D. Barnett. 2010. Parallel Test Execution. (2010). https://patents.google.com/patent/US20120102462A1/en.

[34] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *International Symposium on Foundations of Software Engineering*. 583–594.

[35] Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Symposium on Principles of Programming Languages*. 109–122.

[36] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[37] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Symposium on Principles of Programming Languages*. 357–370.

[38] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*. 643–653.

[39] A. Memon, Zebao Gao, Bao Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. 2017. Taming Google-scale continuous testing. In *International Conference on Software Engineering, Software Engineering in Practice*. 233–242.

[40] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer.

[41] OPAM 2018. OCaml Package Manager. (2018). https://opam.ocaml.org.

[42] Jesper Öqvist, Görel Hedin, and Boris Magnusson. 2016. Extraction-based regression test selection. In *International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 1–10.

[43] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling Regression Testing to Large Software Systems. In *International Symposium on Foundations of Software Engineering*. 241–251.

[44] Frank Pfenning and Christine Paulin-Mohring. 1990. Inductively defined types in the Calculus of Constructions. In *International Conference on Mathematical Foundations of Programming Semantics*. 209–228.

[45] David L. Rager, Warren A. Hunt, and Matt Kaufmann. 2013. A Parallelized Theorem Prover for a Logic with Parallel Execution. In *International Conference on Interactive Theorem Proving*. 435–450.

[46] Vitor Rodrigues and Matthew Fluet. 2015. Surface Effects for Deterministic Parallelism. In *Symposium on Trends in Functional Programming*. ftp://ftp-sop.inria.fr/indes/TFP15/TFP2015_submission_5.pdf

[47] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *Transactions on Software Engineering* 22, 8 (1996), 529–551.

[48] Matthieu Sozeau and Nicolas Tabareau. 2014. Universe Polymorphism in Coq. In *International Conference on Interactive Theorem Proving*. 499–514.

[49] Speeding up the build 2018. Speeding up the build. (2018). http://docs.travis-ci.com/user/speeding-up-the-build.

[50] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively Prioritizing Tests in Development Environment. In *International Symposium on Software Testing and Analysis*. 97–106.

[51] StructTact 2018. StructTact library. (2018). https://github.com/uwplse/StructTact.

[52] Enrico Tassi. 2019. Coq Manual: Asynchronous and Parallel Proof Processing. (2019). https://coq.inria.fr/refman/addendum/parallel-proof-processing.html.

[53] The Coq Development Team. 2018. Coq Manual: Utilities. (2018). https://coq.inria.fr/refman/practical-tools/utilities.html.

[54] TravisCI 2018. Travis CI. (2018). https://travis-ci.org.

[55] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2017).

[56] Makarius Wenzel. 2013. PIDE as front-end technology for Coq. *CoRR* abs/1304.6626 (2013). http://arxiv.org/abs/1304.6626

[57] Makarius Wenzel. 2013. Shared-Memory Multiprocessing for Interactive Theorem Proving. In *International Conference on Interactive Theorem Proving*. 418–434.

[58] Makarius Wenzel. 2014. Asynchronous User Interaction and Tool Integration in Isabelle/PIDE. In *International Conference on Interactive Theorem Proving*. 515–530.

[59] Makarius Wenzel. 2017. Scaling Isabelle Proof Document Processing. (December 2017). http://sketis.net/wp-content/uploads/2017/12/Isabelle_Scaling_Dec-2017.pdf.

[60] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Conference on Programming Language Design and Implementation*. 357–368.

[61] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for change in a formal verification of the Raft consensus protocol. In *Conference on Certified Programs and Proofs*. 154–165.

[62] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[63] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *International Conference on Software Maintenance*. 23–32.