piCoq: Parallel Regression Proving for Large-Scale Verification Projects

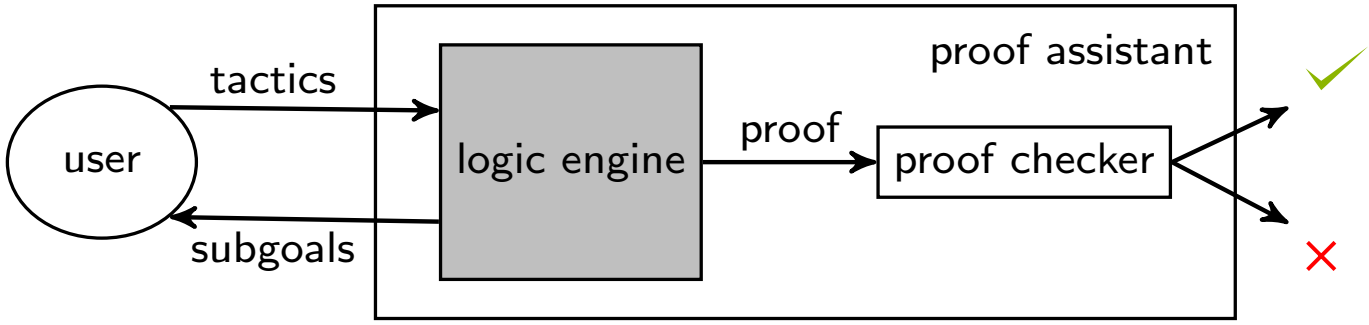
Karl Palmskog, Ahmet Celik, and Milos Gligoric

The University of Texas at Austin, USA

THE UNIVERSITY OF TEXAS AT AUSTIN

Verification Using Proof Assistants

1. encode definitions in (higher-order) formalism
2. prove propositions **interactively** using powerful **tactics**
3. check soundness of every low-level step



examples: Coq, HOL4, HOL Light, Isabelle/HOL, Lean, Nuprl, ...

Some Large-Scale Proof Assistant Projects

Project	**Year**	**Assistant**	**Check Time**	**LOC**
4-Color Theorem	2005	Coq	hours	60k
Odd Order Theorem	2012	Coq	hours	150k
Kepler Conjecture	2015	HOL Light	days	500k
CompCert C compiler	2009	Coq	tens of mins	40k
Cogent (BilbyFS)	2016	Isabelle/HOL	hours	14k
Verdi (Raft consensus)	2016	Coq	tens of mins	50k

problem: long proof checking times

Proof Engineering Techniques For Effective Proving

Proof selection: check only proofs affected by changes

- file/module selection
- proof selection

Examples: Make, Isabelle [ITP '14], iCoq [ASE '17]

Proof Engineering Techniques For Effective Proving

Proof selection: check only proofs affected by changes

- file/module selection
- proof selection

Examples: Make, Isabelle [ITP '14], iCoq [ASE '17]

Proof parallelization: leverage multi-core hardware

- parallel checking of proofs
- parallel checking of files

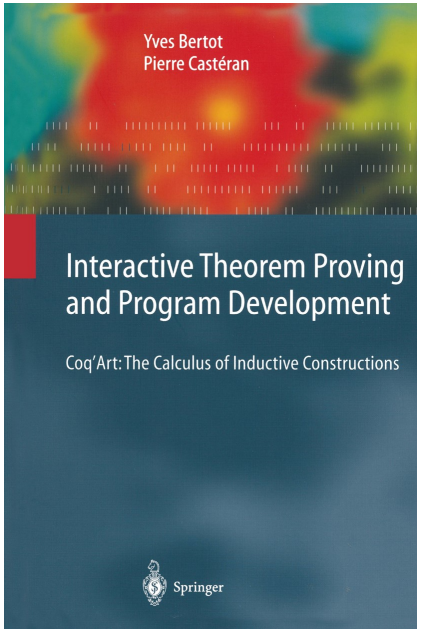
Examples: Make, Isabelle [ITP '13], Coq [ITP '15]

Our Contributions

- **taxonomy** of regression proving techniques that leverage both selection and parallelism
- **implementation** of techniques in tool, piCoq, that supports Coq projects (useful for CI, e.g., Travis on GitHub)
- **evaluation** using piCoq on six open source projects (23 kLOC over 22 revisions per project, on average)

The Coq Proof Assistant (1985-present)

- based on constructive dependent type theory
- Gallina – programming/specification language
- Ltac – proof tactic language
- small trusted core checker for programs & proofs



Coq Source File Example

```
Require Import List.
Require Import ListUtil.

Import ListNotations.

Fixpoint dedup A 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 :
 forall 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

Coq Source File Example

```
Require Import List.
Require Import ListUtil.

Import ListNotations.

Fixpoint dedup A 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 :
 forall 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` statements expressing file dependencies.

Coq Source File Example

```
Require Import List.
Require Import ListUtil.

Import ListNotations.

Fixpoint dedup A 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 :
 forall 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

Definition of a recursive function to remove duplicate list elements in Gallina.

Coq Source File Example

```
Require Import List.
Require Import ListUtil.

Import ListNotations.

Fixpoint dedup A 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 :
 forall 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

Statement (type) of a lemma in Gallina.

Coq Source File Example

```
Require Import List.
Require Import ListUtil.

Import ListNotations.

Fixpoint dedup A 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 :
 forall 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

Proof script in Ltac – potentially time-consuming to process.

Coq Proof-Checking Toolchain

Legacy Top-Down Proof Checking (1990s)

- `coqc`: compilation of source `.v` files to binary `.vo` files
- `.vo` files contain **functions and all proofs**
- file-level parallelism via Make

Coq Proof-Checking Toolchain

Legacy Top-Down Proof Checking (1990s)

- `coqc`: compilation of source `.v` files to binary `.vo` files
- `.vo` files contain **functions and all proofs**
- file-level parallelism via Make

Quick Compilation and Asynchronous Checking (2015)

- `coqc -quick`: compilation of `.v` files to binary `.vio` files
- `.vio` files contain **functions and proof tasks**
- proof tasks checkable asynchronously in parallel

Regression Proving Modes for Coq (Taxonomy)

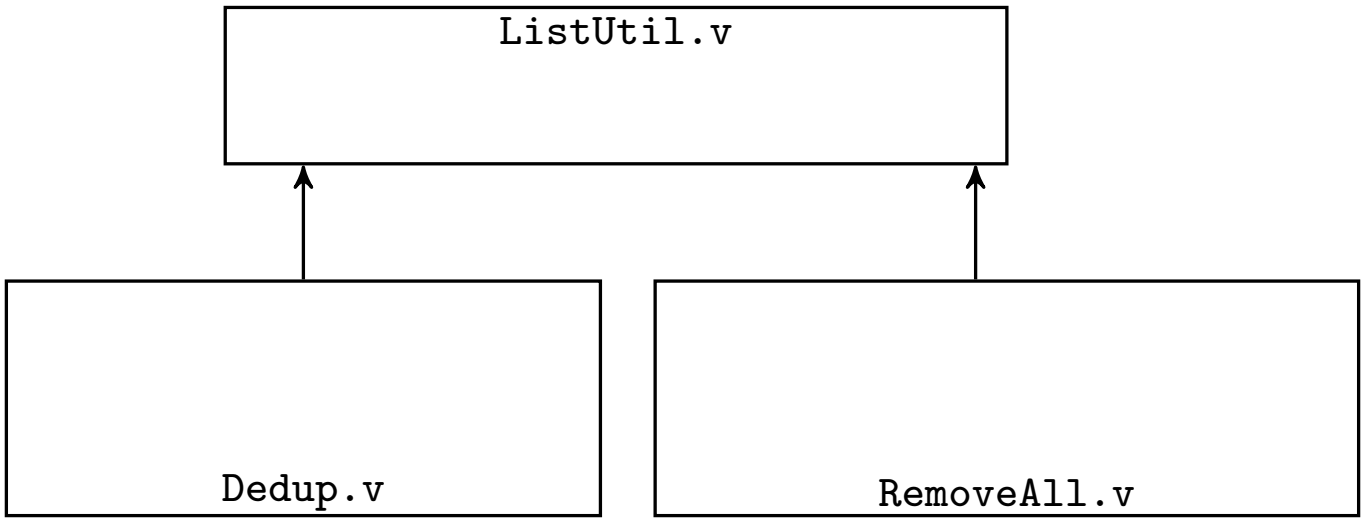
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	N/A
Proof level	`p·none`	`p·file`	`p·icoq`

f·none Mode: File-Level Parallelization, No Selection

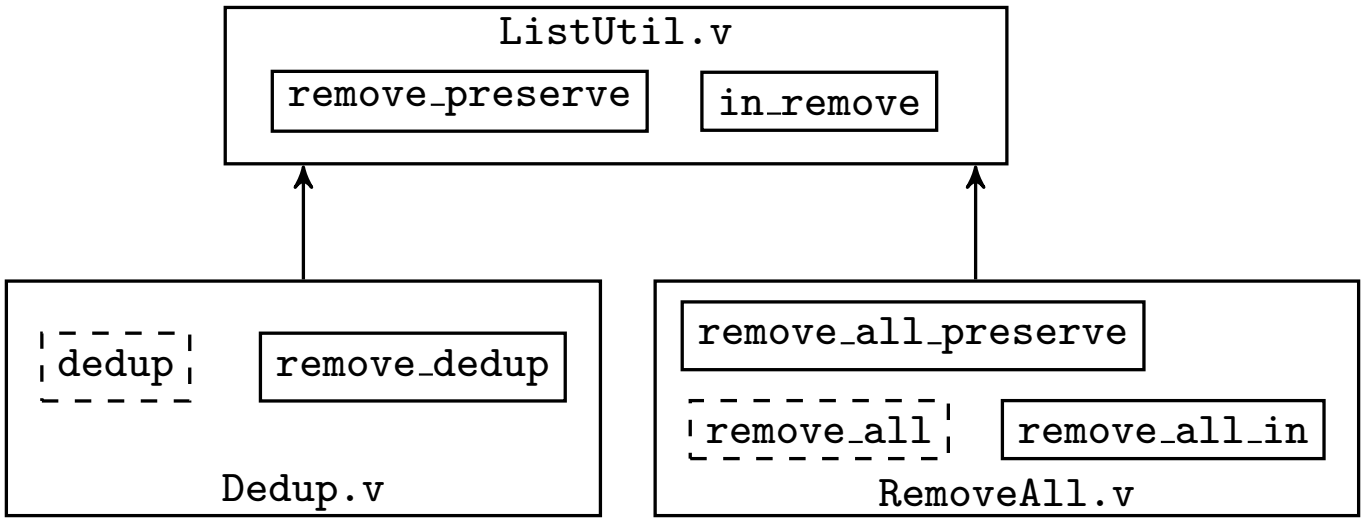
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	f·none	f·file	N/A
Proof level	p·none	p·file	p·icoq

- legacy mode used in most GitHub Coq projects
- no overhead from proof task management or dep. tracking
- parallelism restricted by file dependency graph
- overhead from writing proofs to disk

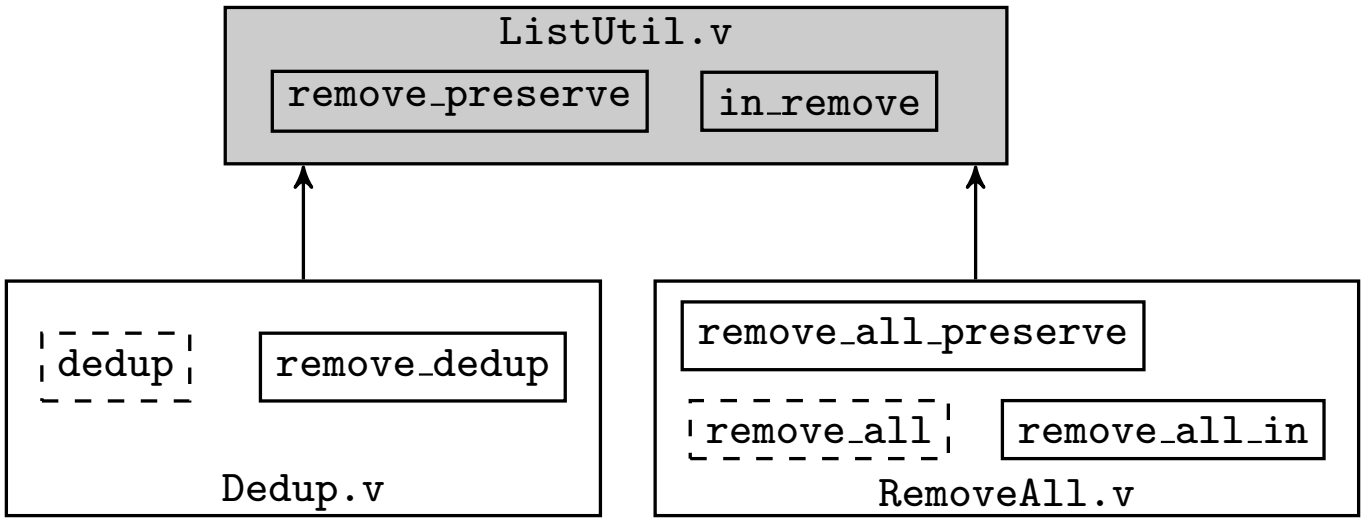
f•none Mode in Practice



f·none Mode in Practice

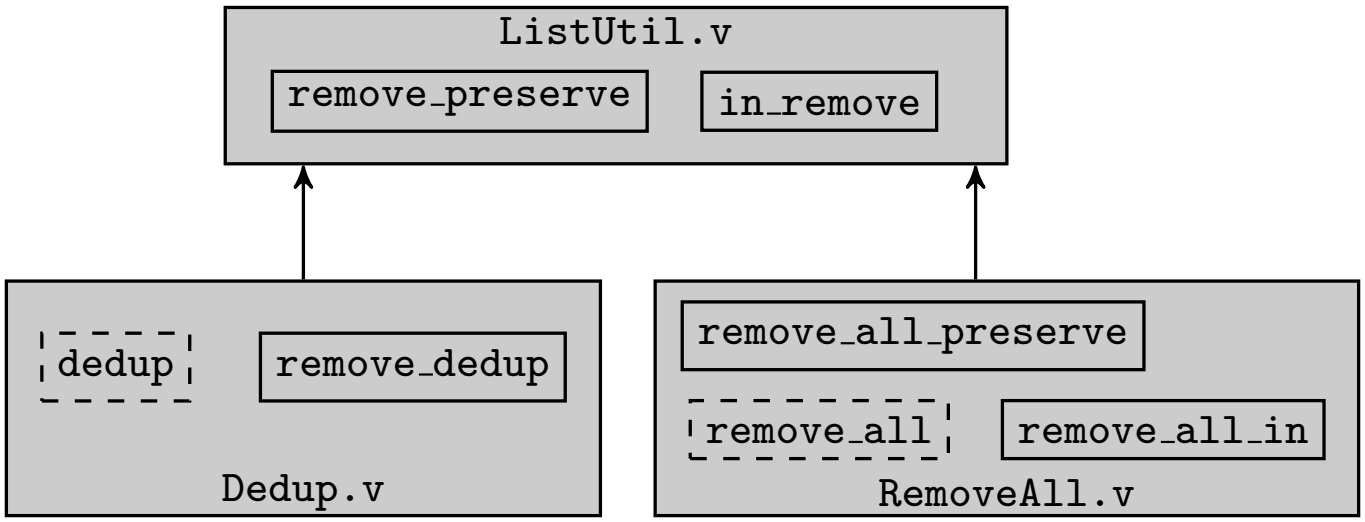


f·none Mode in Practice



Phase	Task	Definitions and Lemmas
1	ListUtil.vo	remove_preserve, in_remove

f•none Mode in Practice



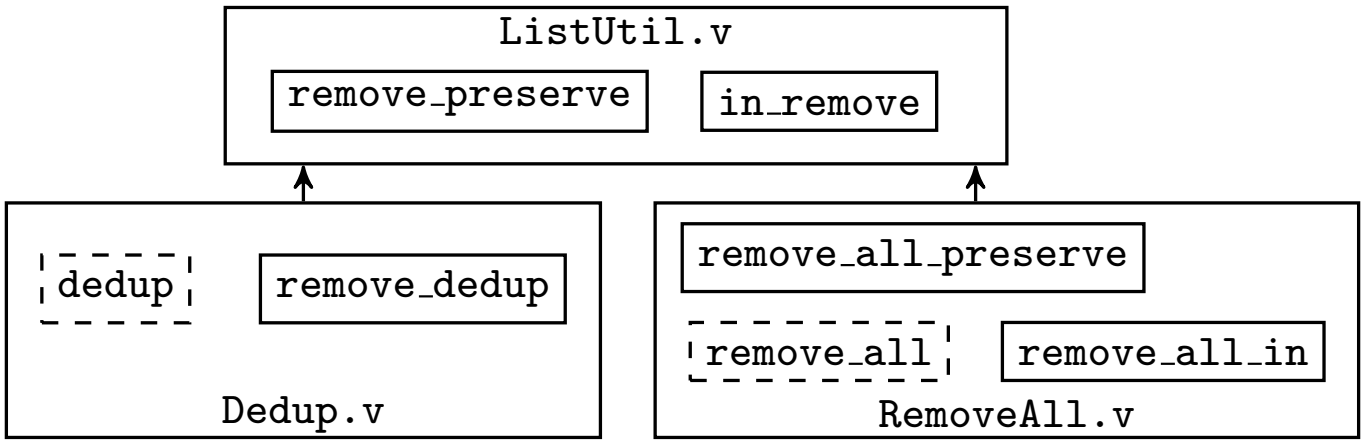
Phase	Task	Definitions and Lemmas
1	ListUtil.vo	remove_preserve, in_remove
2	Dedup.vo	dedup, remove_dedup
2	RemoveAll.vo	remove_all, remove_all_in, remove_all_preserve

p·none Mode: Proof-Level Parallelization, No Selection

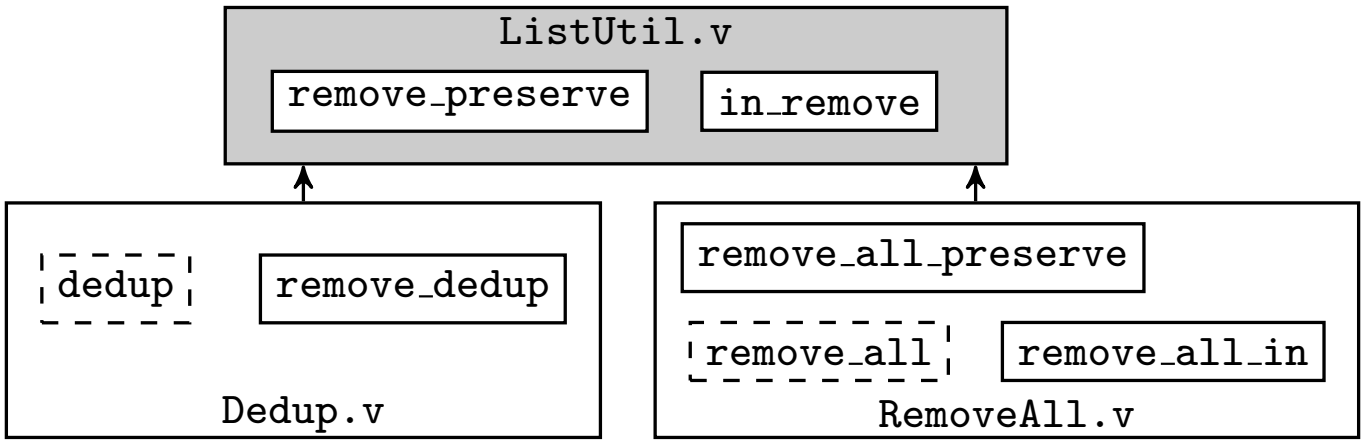
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	f·none	f·file	N/A
Proof level	p·none	p·file	p·icoq

- legacy mode used in some GitHub Coq projects
- overhead from proof task management
- parallelism (largely) unrestricted by file dependency graph
- no overhead from writing proofs to disk and dep. tracking

p·none Mode in Practice

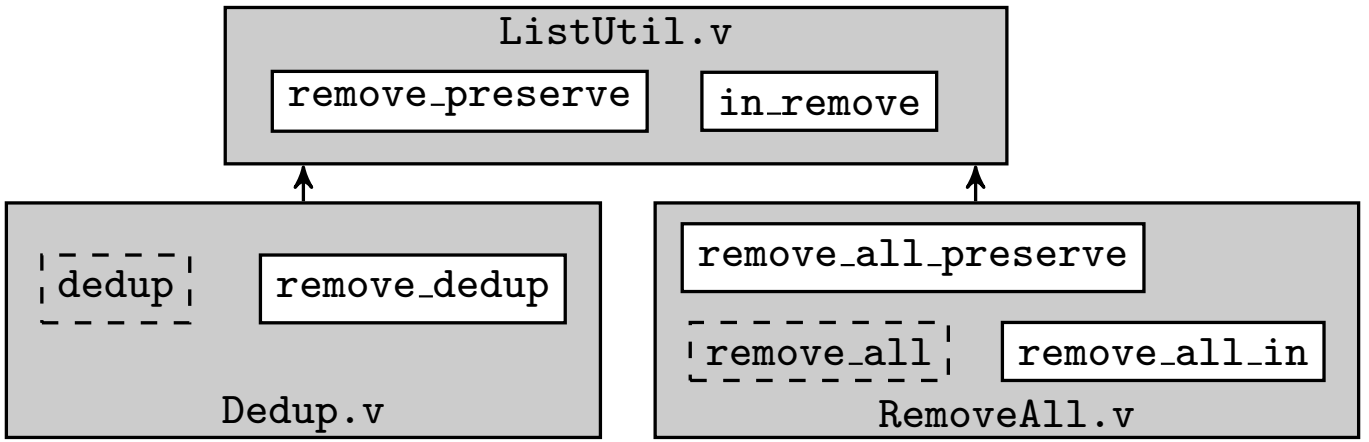


p·none Mode in Practice



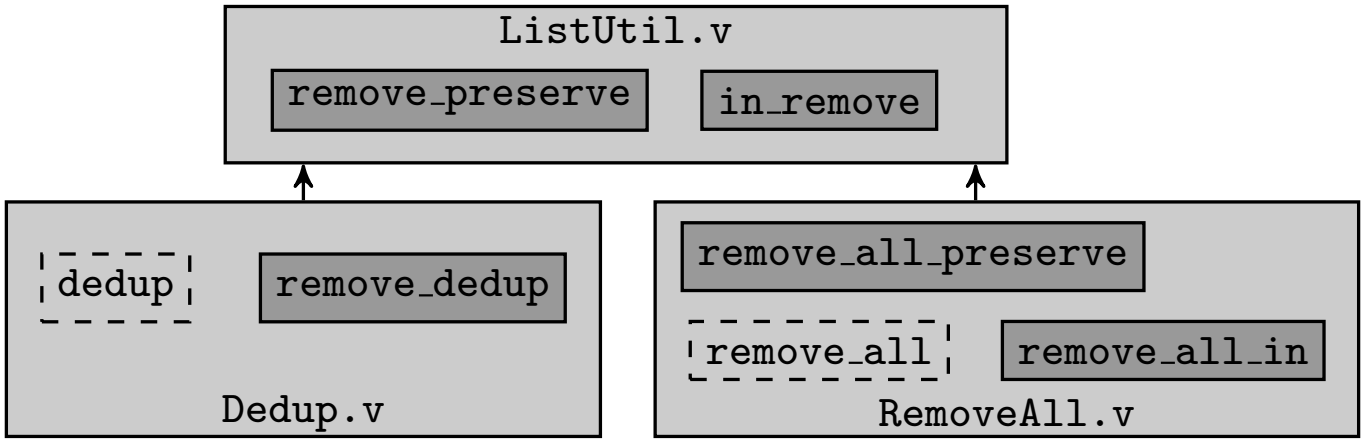
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~

p·none Mode in Practice



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~~

p·none Mode in Practice



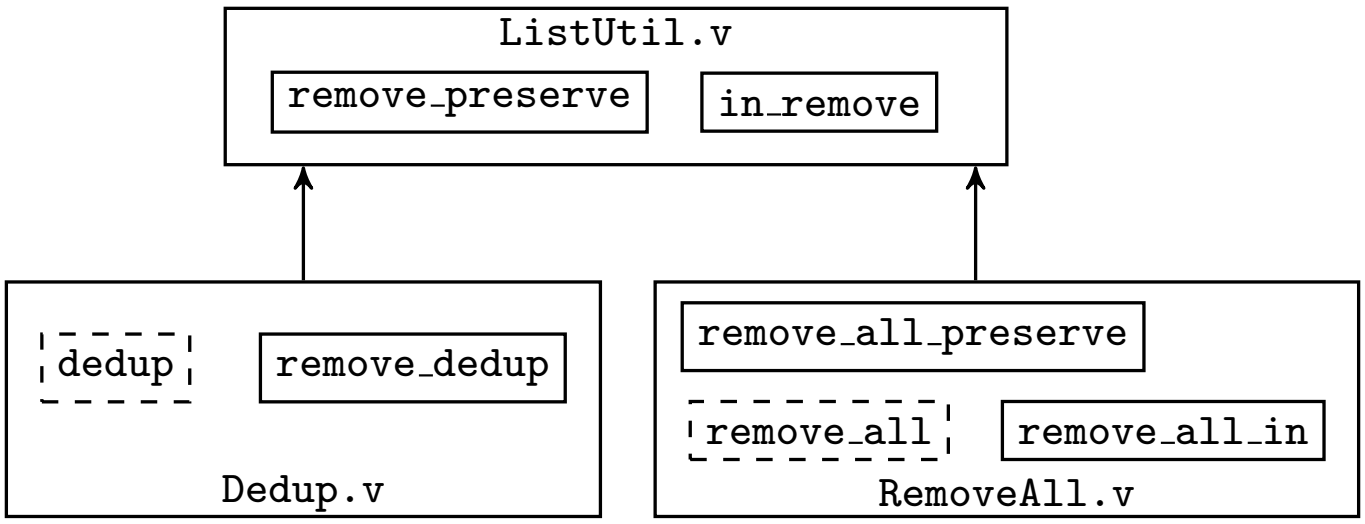
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~
2	Dedup.vio	dedup, ~~remove_dedup~~
2	RemoveAll.vio	remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~
3	checking	remove_preserve
3	checking	in_remove
3	checking	remove_dedup
3	checking	remove_all_in
3	checking	remove_all_preserve

`f·file` Mode: File-Level Parallelization, File Selection

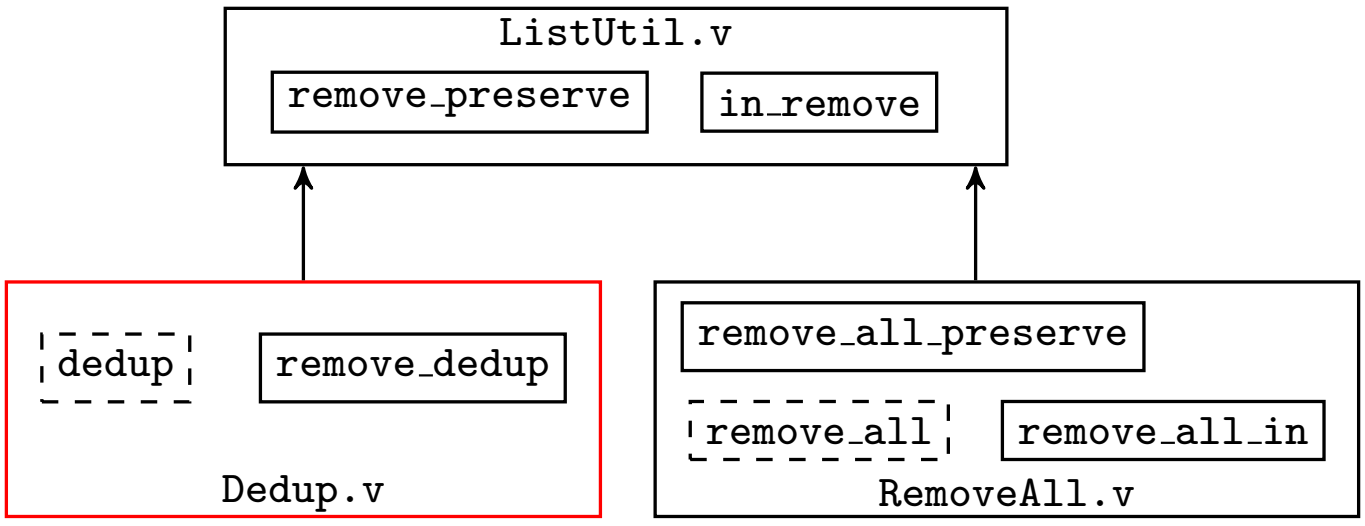
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	N/A
Proof level	`p·none`	`p·file`	`p·icoq`

- novel mode that persists file checksums
- overhead from file dependency tracking
- parallelism restricted by file dependency graph
- overhead from writing proofs to disk

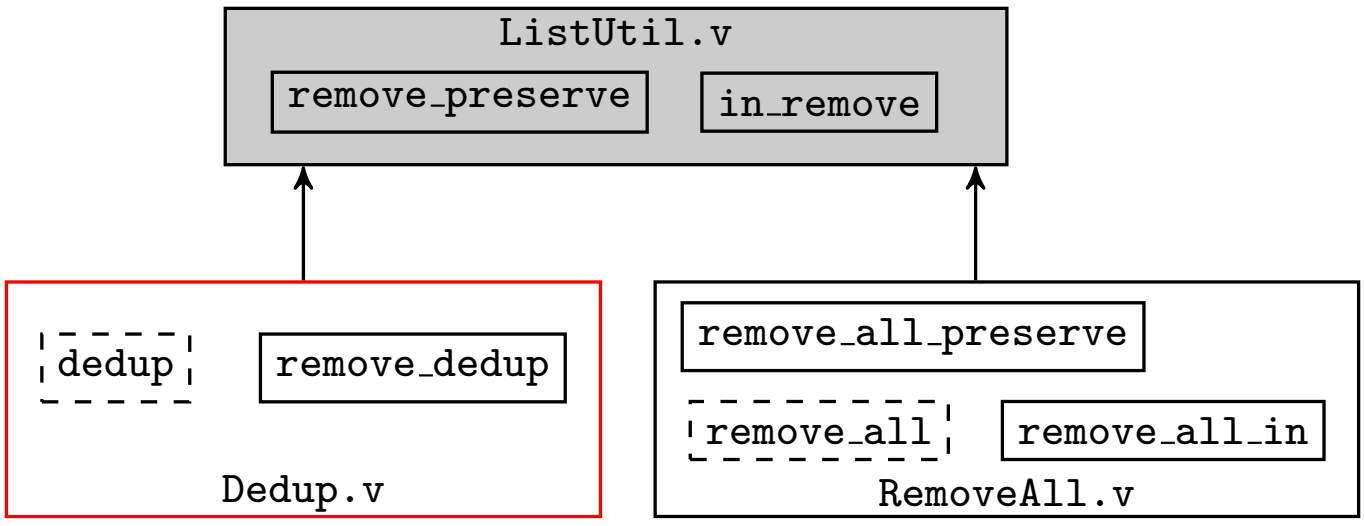
`f·file` Mode in Practice



`f·file` Mode in Practice

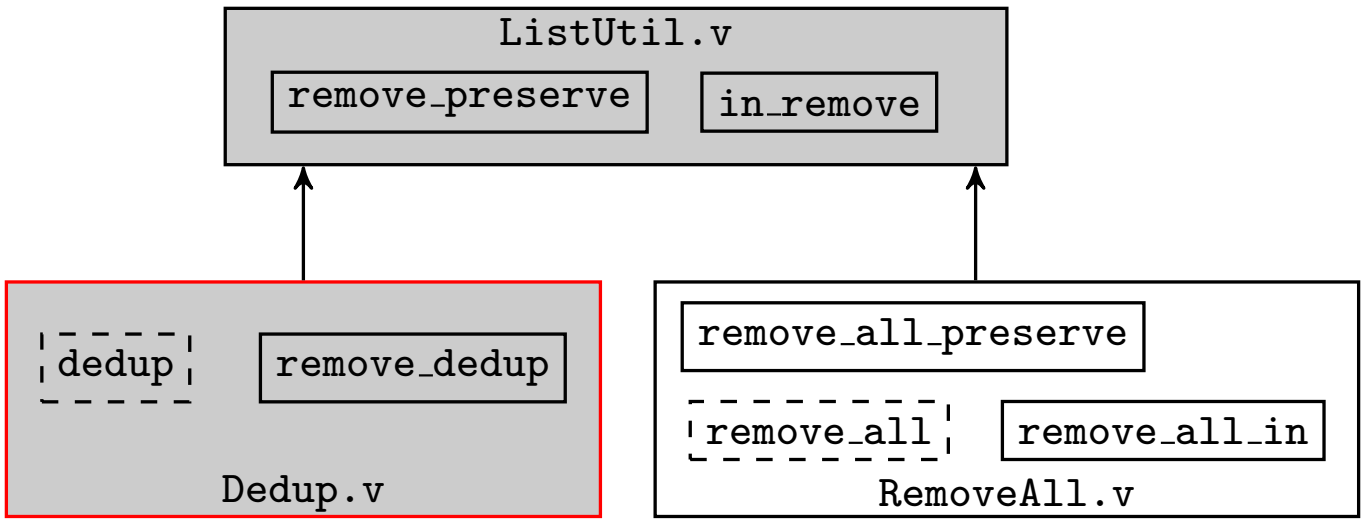


`f·file` Mode in Practice



Phase	Task	Definitions and Lemmas
1	`ListUtil.vo`	`remove_preserve`, `in_remove`

`f·file` Mode in Practice



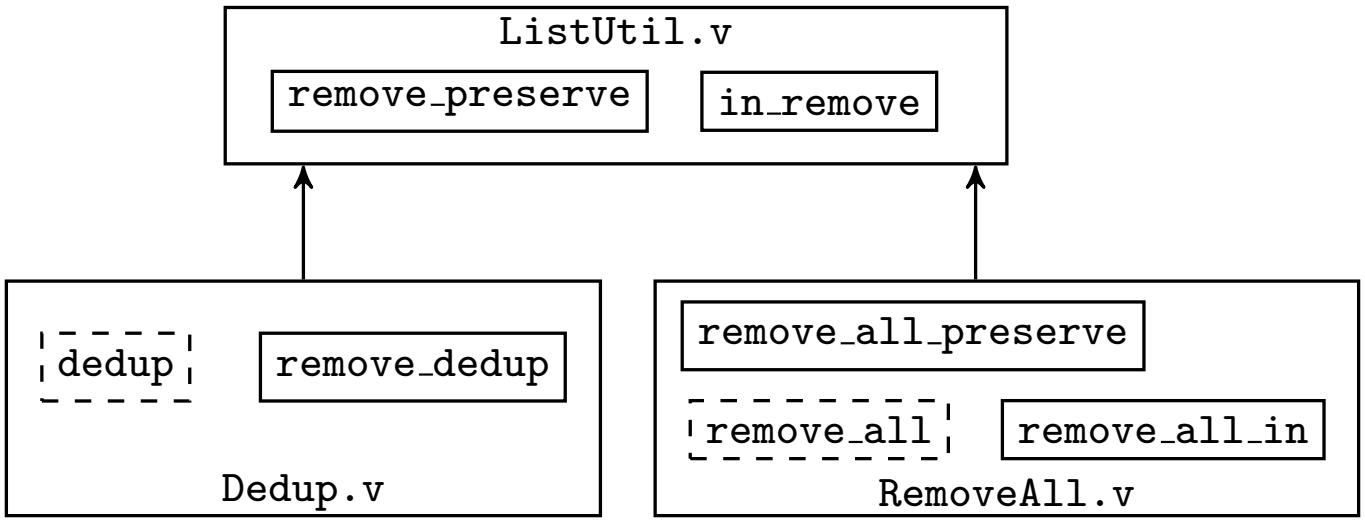
Phase	Task	Definitions and Lemmas
1	`ListUtil.vo`	`remove_preserve`, `in_remove`
2	`Dedup.vo`	`dedup`, `remove_dedup`

`p·file` Mode: Proof-Level Parallelism, File Selection

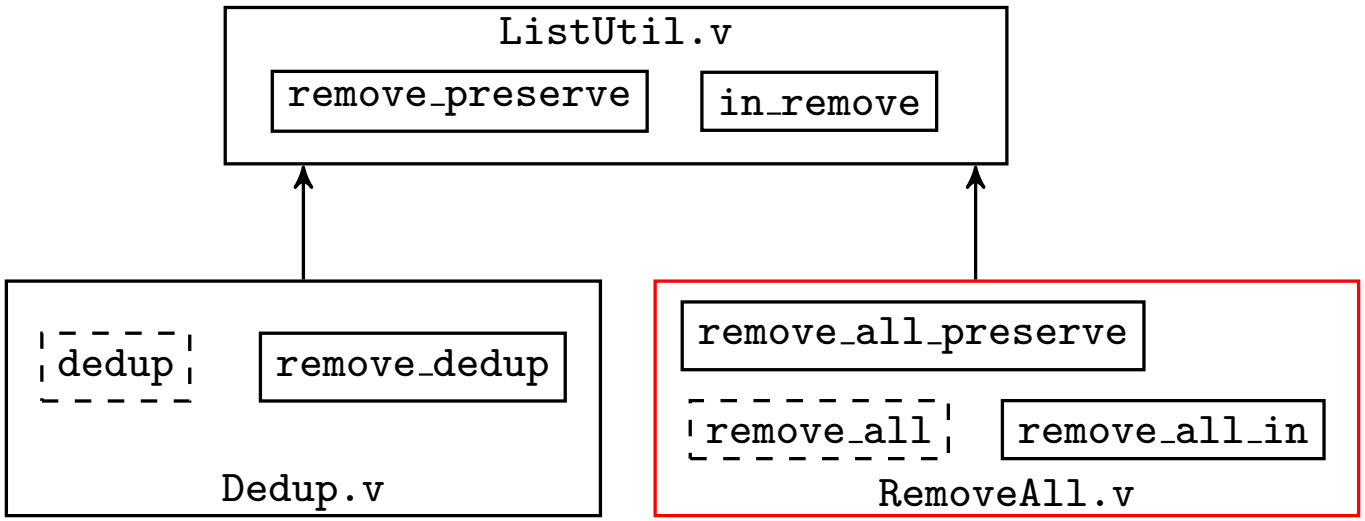
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	N/A
Proof level	`p·none`	`p·file`	`p·icoq`

- novel mode that persists file checksums
- overhead from file dependency tracking
- parallelism (mostly) unrestricted by file dependency graph
- no overhead from writing proofs to disk

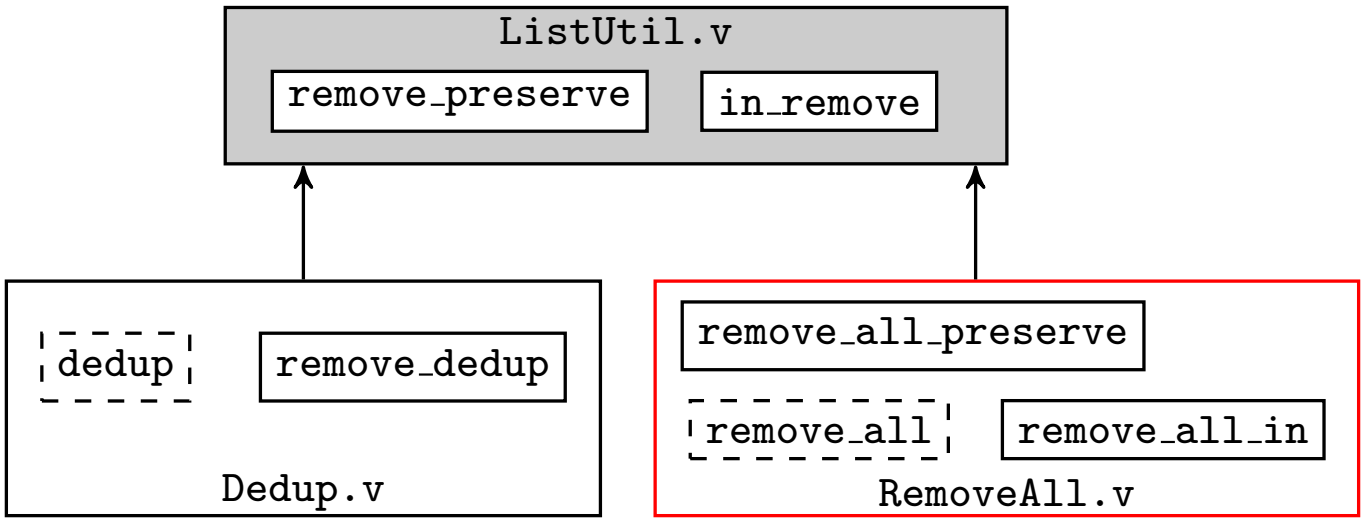
p·file Mode in Practice



`p·file` Mode in Practice

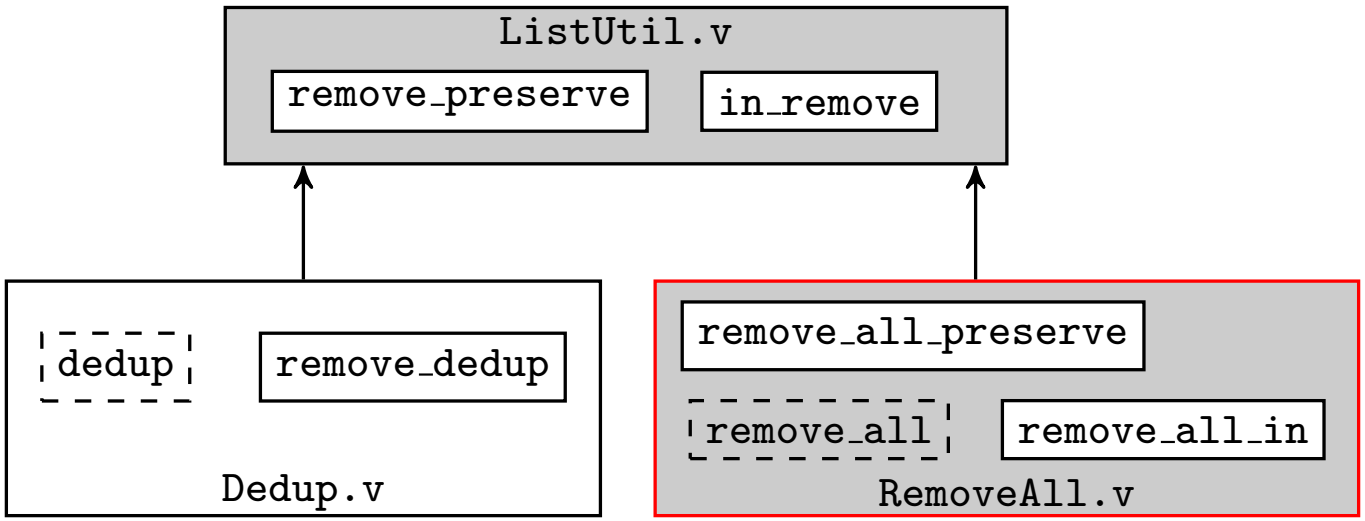


p·file Mode in Practice



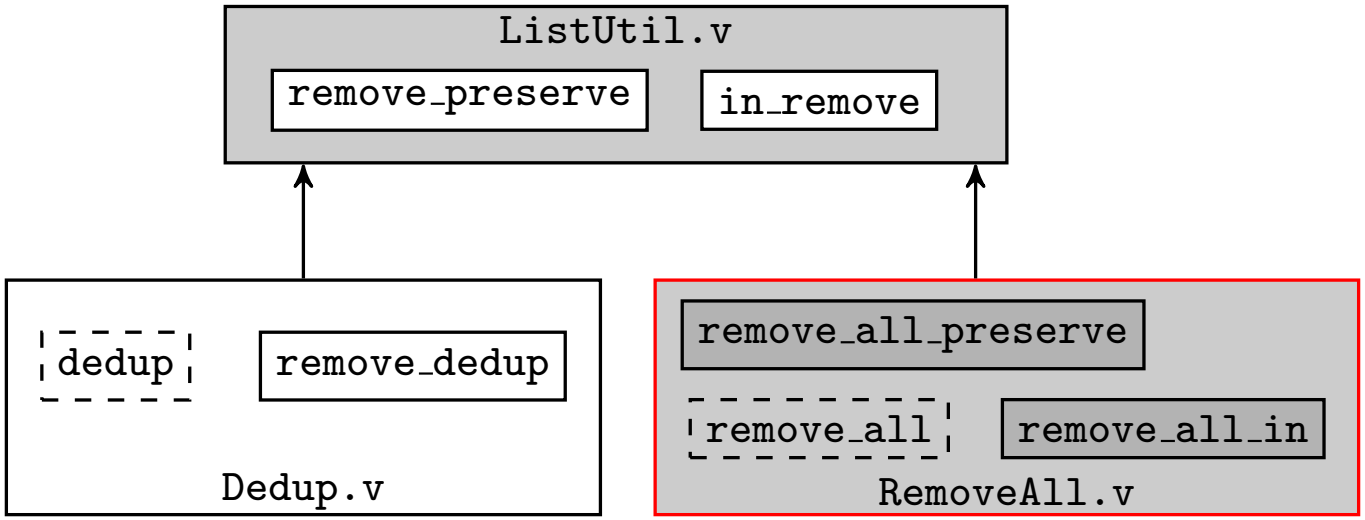
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~

p·file Mode in Practice



Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~
2	RemoveAll.vio	remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~

p·file Mode in Practice



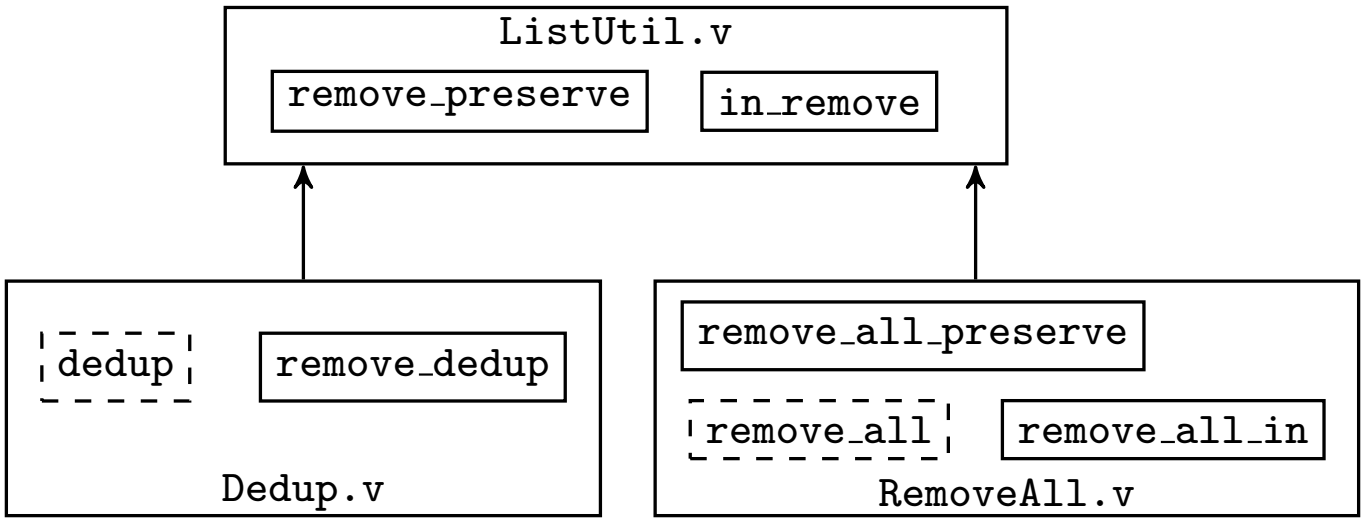
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~
2	RemoveAll.vio	remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~
3	checking	remove_all_in
3	checking	remove_all_preserve

`p·icoq` Mode: Proof-Level Parallelism, Proof Selection

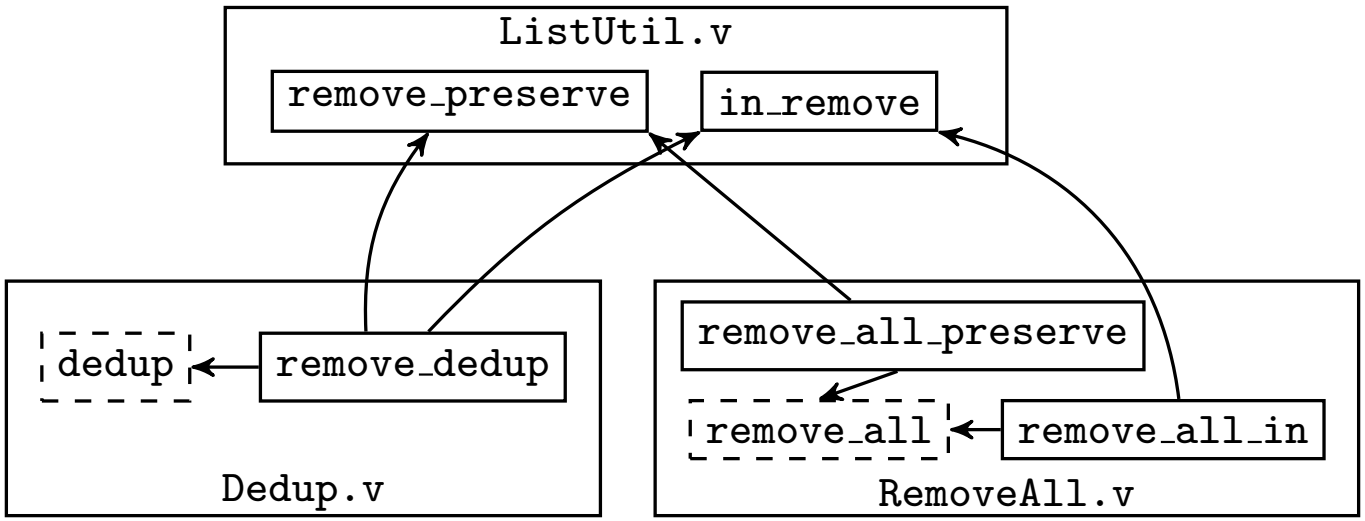
Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	N/A
Proof level	`p·none`	`p·file`	`p·icoq`

- novel mode that persists file & proof checksums
- overhead from file & proof dependency tracking
- parallelism (mostly) unrestricted by file dependency graph
- no overhead from writing proofs to disk

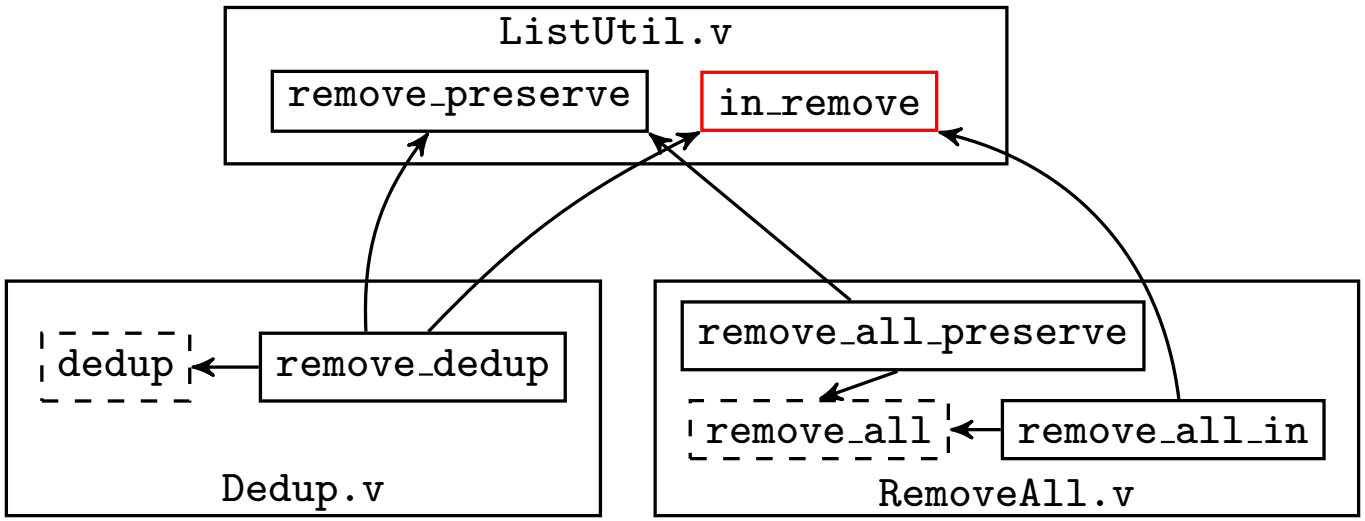
p·icoq Mode in Practice



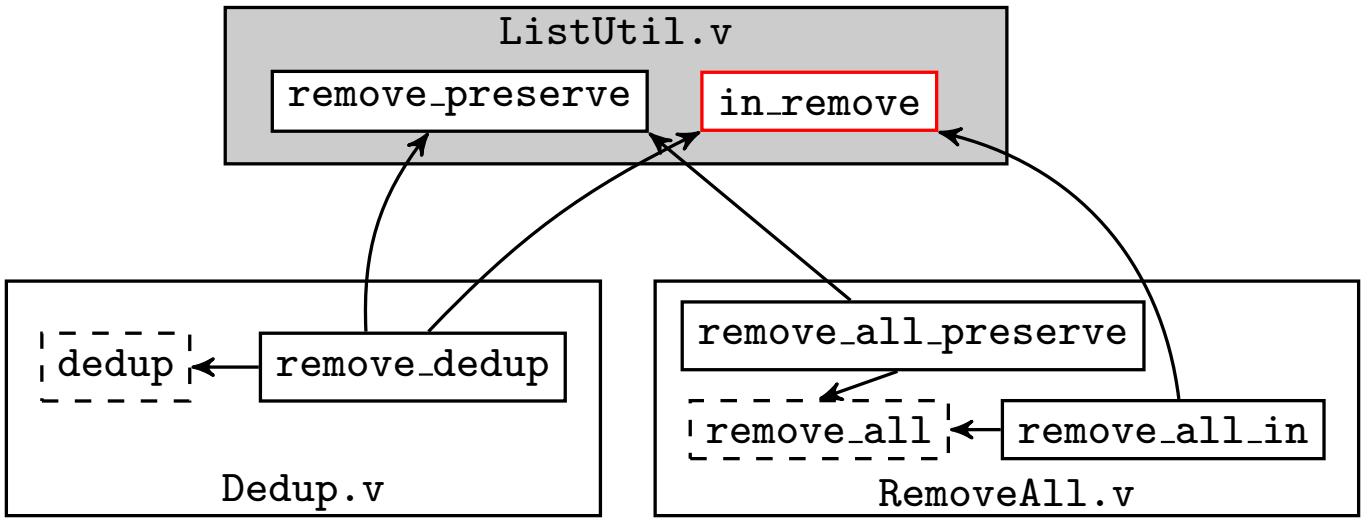
p·icoq Mode in Practice



p·icoq Mode in Practice

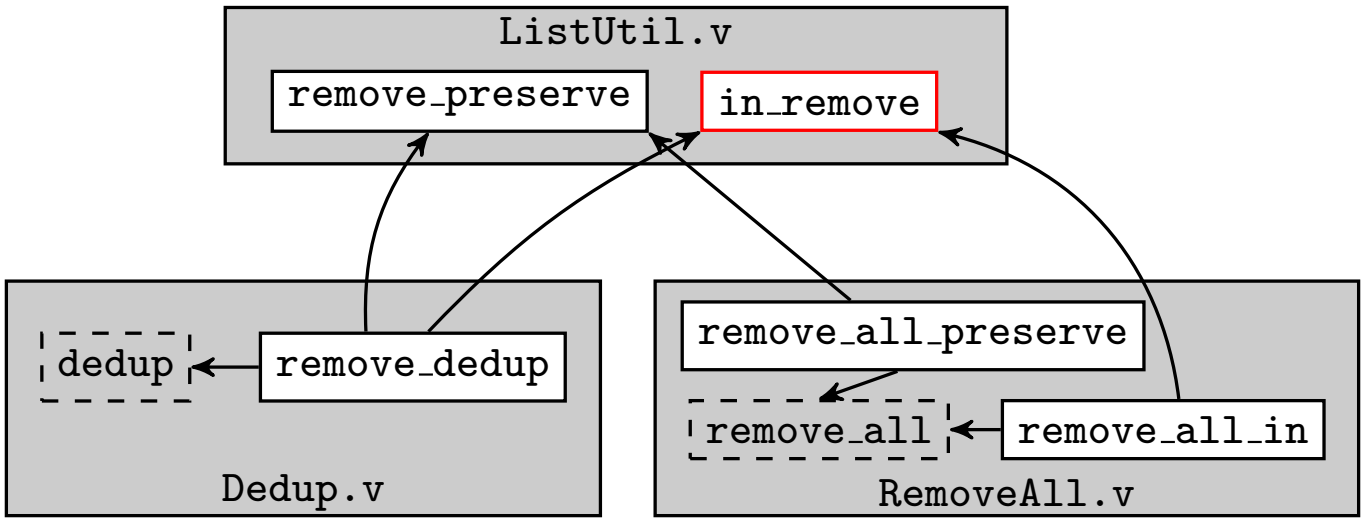


p·icoq Mode in Practice



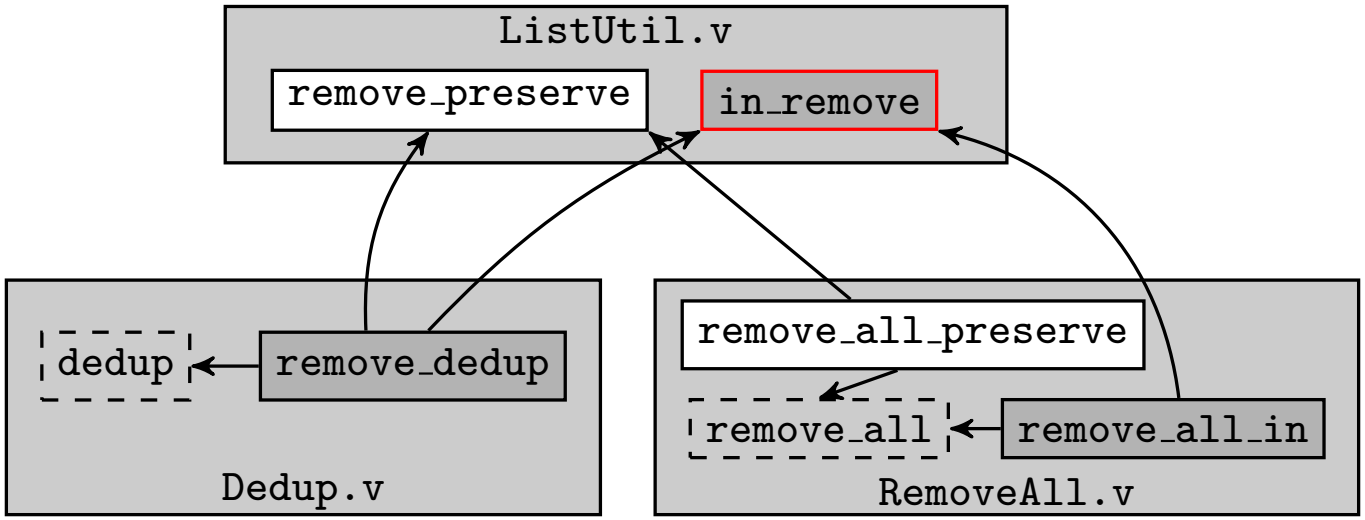
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~

p·icoq Mode in Practice



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~~

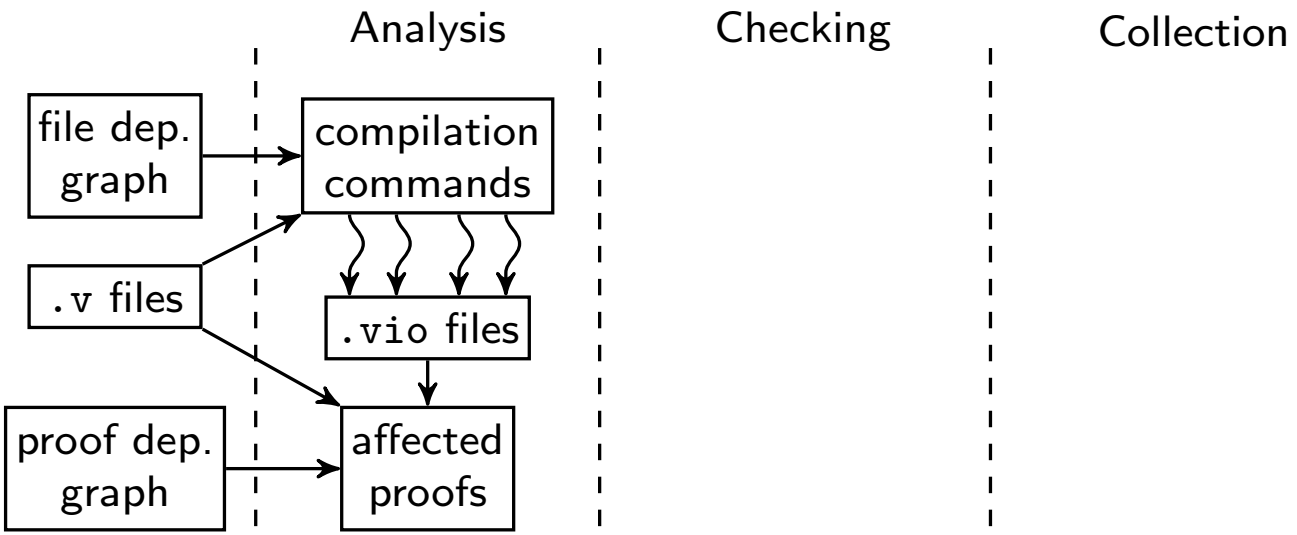
p·icoq Mode in Practice



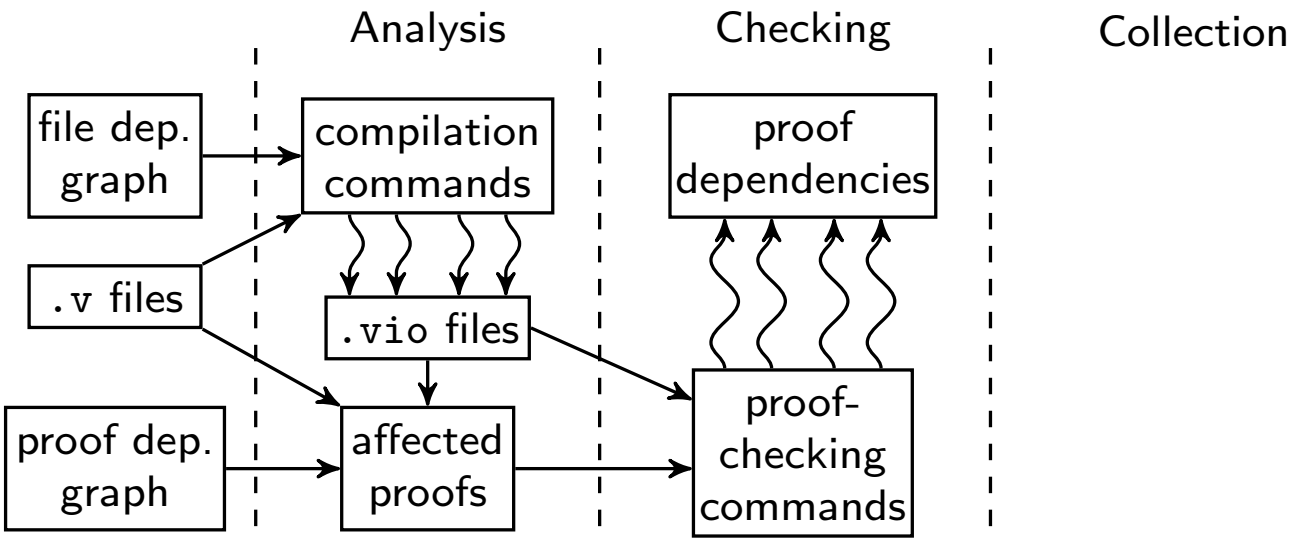
Phase	Task	Definitions and Lemmas
1	ListUtil.vio	~~remove_preserve~~, ~~in_remove~~
2	Dedup.vio	dedup, ~~remove_dedup~~
2	RemoveAll.vio	remove_all, ~~remove_all_in~~, ~~remove_all_preserve~~
3	checking	in_remove
3	checking	remove_dedup
3	checking	remove_all_in

p·icoq Workflow with 4-way Parallelization

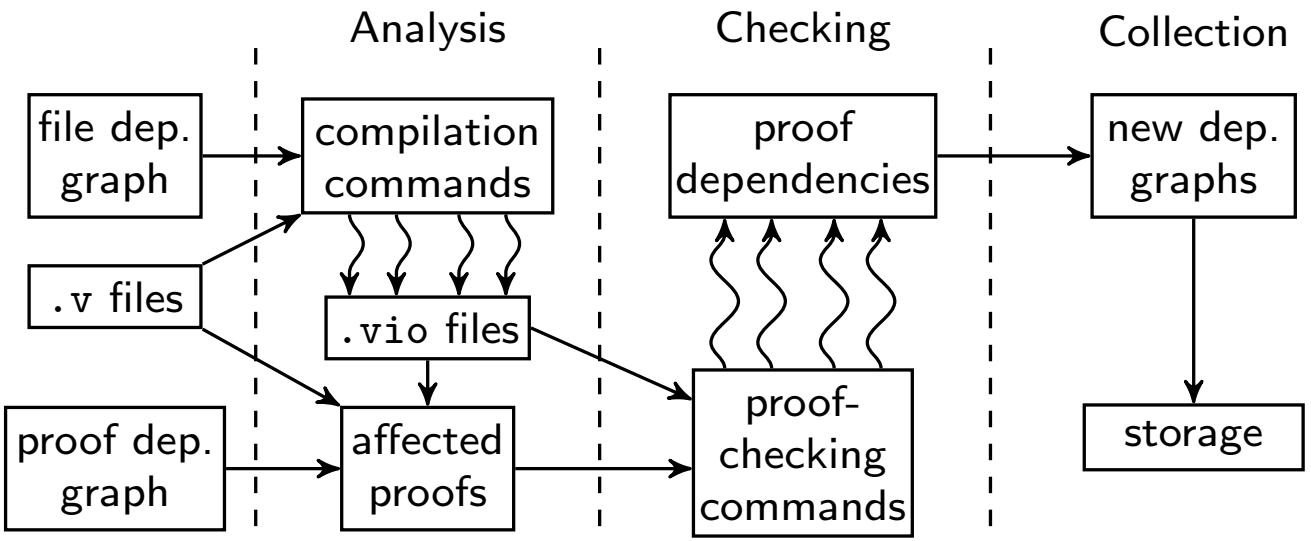
p·icoq Workflow with 4-way Parallelization



p·icoq Workflow with 4-way Parallelization



p·icoq Workflow with 4-way Parallelization



piCoq Tool Implementation

- extension of iCoq toolchain (Java, OCaml, bash)
- Java task executor for parallel compilation of `.vo`/`.vio` files
- extension of `coqc` for parallel checking of (and dependency extraction from) specific proofs across files

```
$ coqc -schedule-vio-task-depends-checking 4 \
   file1.vio 1,15,16 \
   file2.vio 3,10,11,13,20
```

piCoq Tool Implementation

- extension of iCoq toolchain (Java, OCaml, bash)
- Java task executor for parallel compilation of `.vo`/`.vio` files
- extension of coqc for parallel checking of (and dependency extraction from) specific proofs across files

```
$ coqc -schedule-vio-task-depends-checking 4 \
   file1.vio 1,15,16 \
   file2.vio 3,10,11,13,20
```

piCoq Tool Implementation

- extension of iCoq toolchain (Java, OCaml, bash)
- Java task executor for parallel compilation of `.vo`/`.vio` files
- extension of coqc for parallel checking of (and dependency extraction from) specific proofs across files

```
$ coqc -schedule-vio-task-depends-checking 4 \
    file1.vio 1,15,16 \
    file2.vio 3,10,11,13,20
```

Evaluation: Open Source Git-Based Projects

Project	**LOC**	**Domain**
Coquelicot	38260	real number analysis
Finmap	5661	finite sets and maps
Flocq	24786	floating-point arithmetic
Fomegac	2637	formal system metatheory
Surface Effects	9621	functional programming languages
Verdi	56147	distributed systems
$\sum$	137112	
Avg.	22852.00	

Evaluation: Open Source Git-Based Projects

Project	**LOC**	**#Revs.**	**#Files**	**#Proof Tasks**
Coquelicot	38260	24	29	1660
Finmap	5661	23	4	959
Flocq	24786	23	40	943
Fomegac	2637	14	13	156
Surface Effects	9621	24	15	289
Verdi	56147	24	222	2756
$\sum$	137112	132	323	6763
Avg.	22852.00	22.00	53.83	1127.16

Evaluation Details

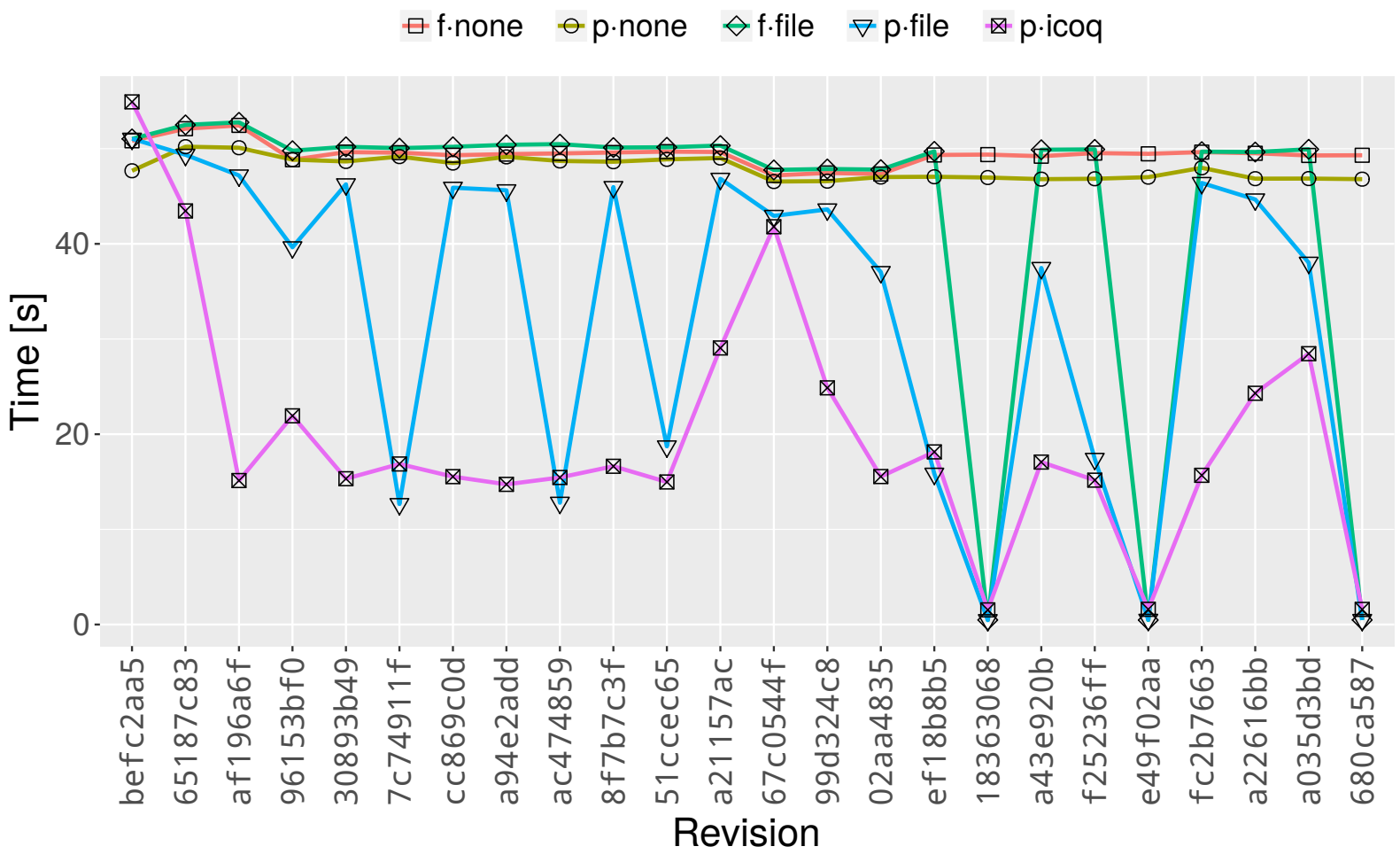
Experiment machine: Intel Core i7-6700 CPU @ 3.40GHz

- 4 CPU cores
- 16 GB memory
- Ubuntu Linux 17.04
- Coq 8.5

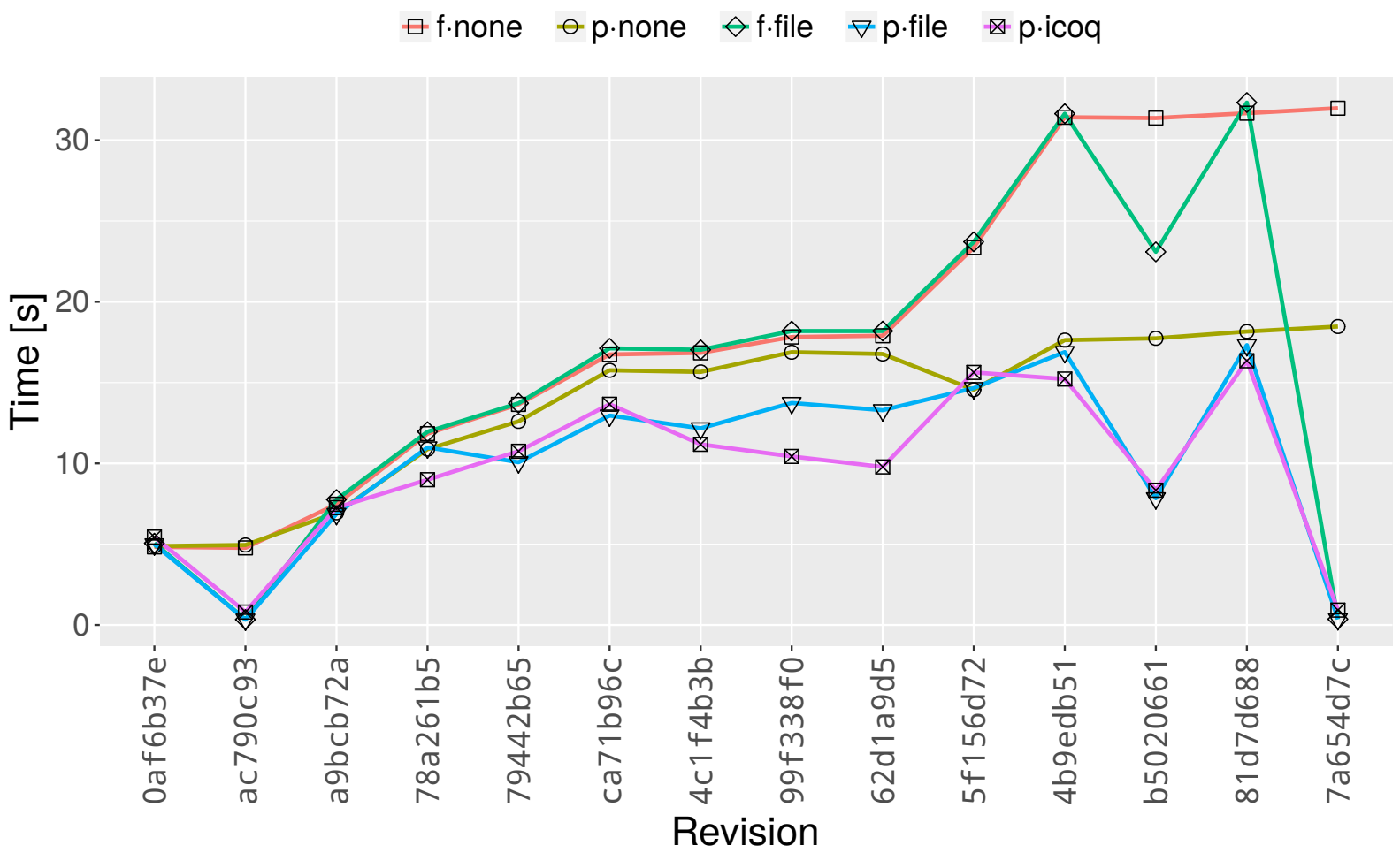
Evaluation Setup

- every build starts from scratch (version control)
- up to 4 parallel jobs/processes
- dependency metadata persisted between revisions

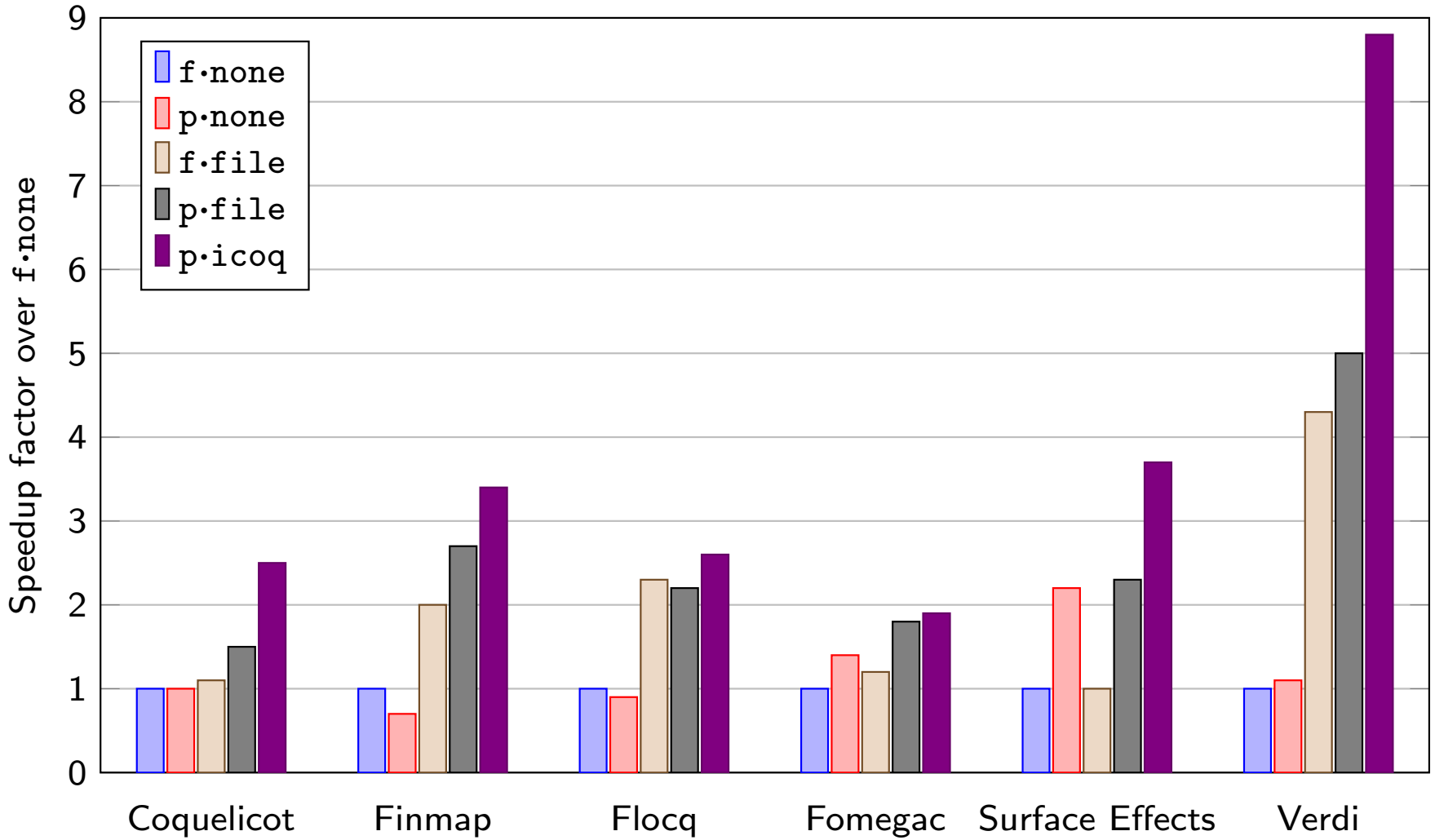
Results with 4-way Parallelization: Coquelicot



Results with 4-way Parallelization: Fomegac

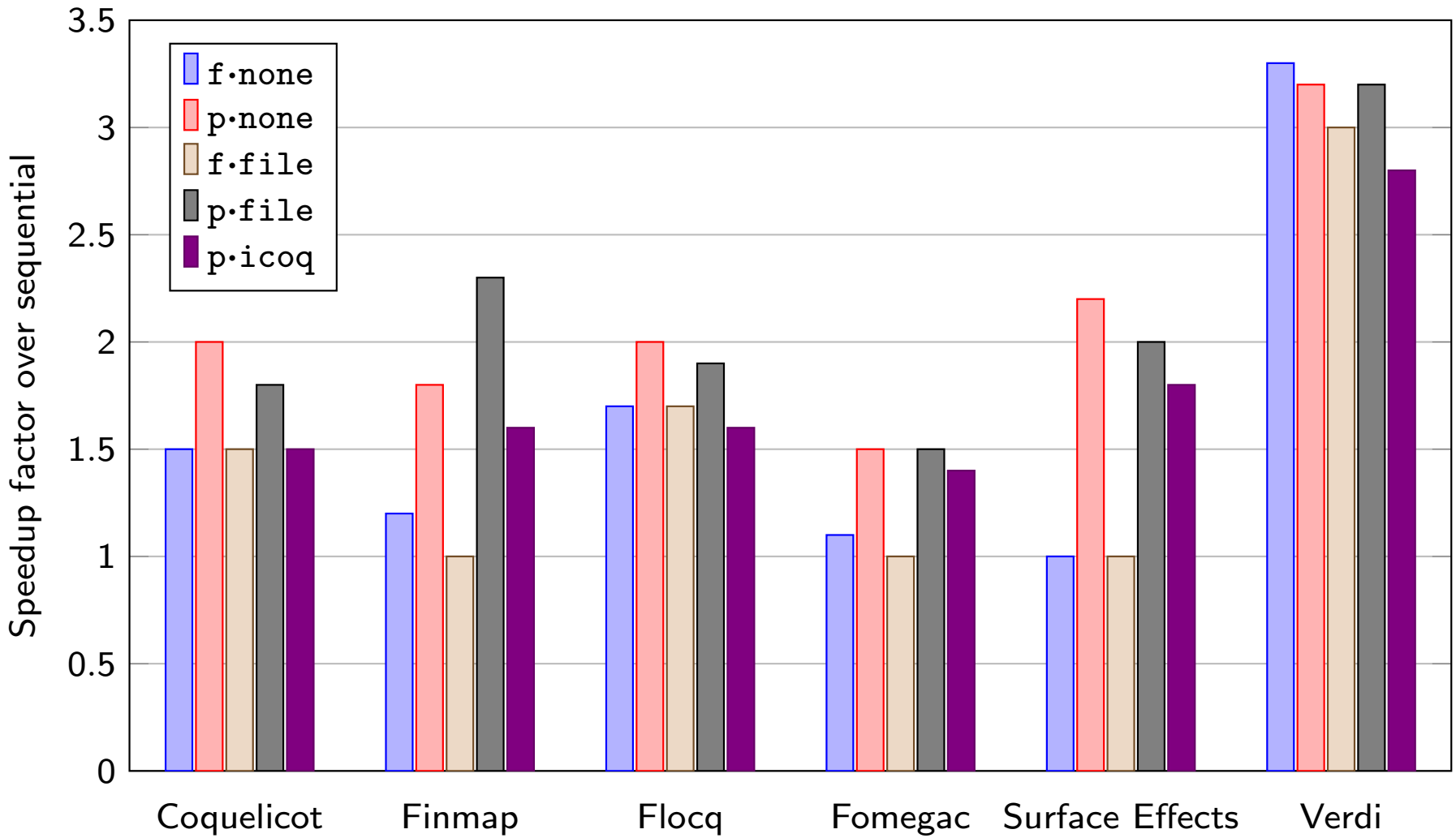


Speedups over `f·none` for 4-way Parallel Checking



"How much faster modes are than the default mode, for each project"

Speedups from Sequential to 4-way Parallel Checking



"Effect of parallelism on each mode and project"

Conclusion

- taxonomy of regression proving modes
- implementation of modes for Coq in tool piCoq
- eval shows speedups for parallelism/selection (up to 28.6×)

Contact us:

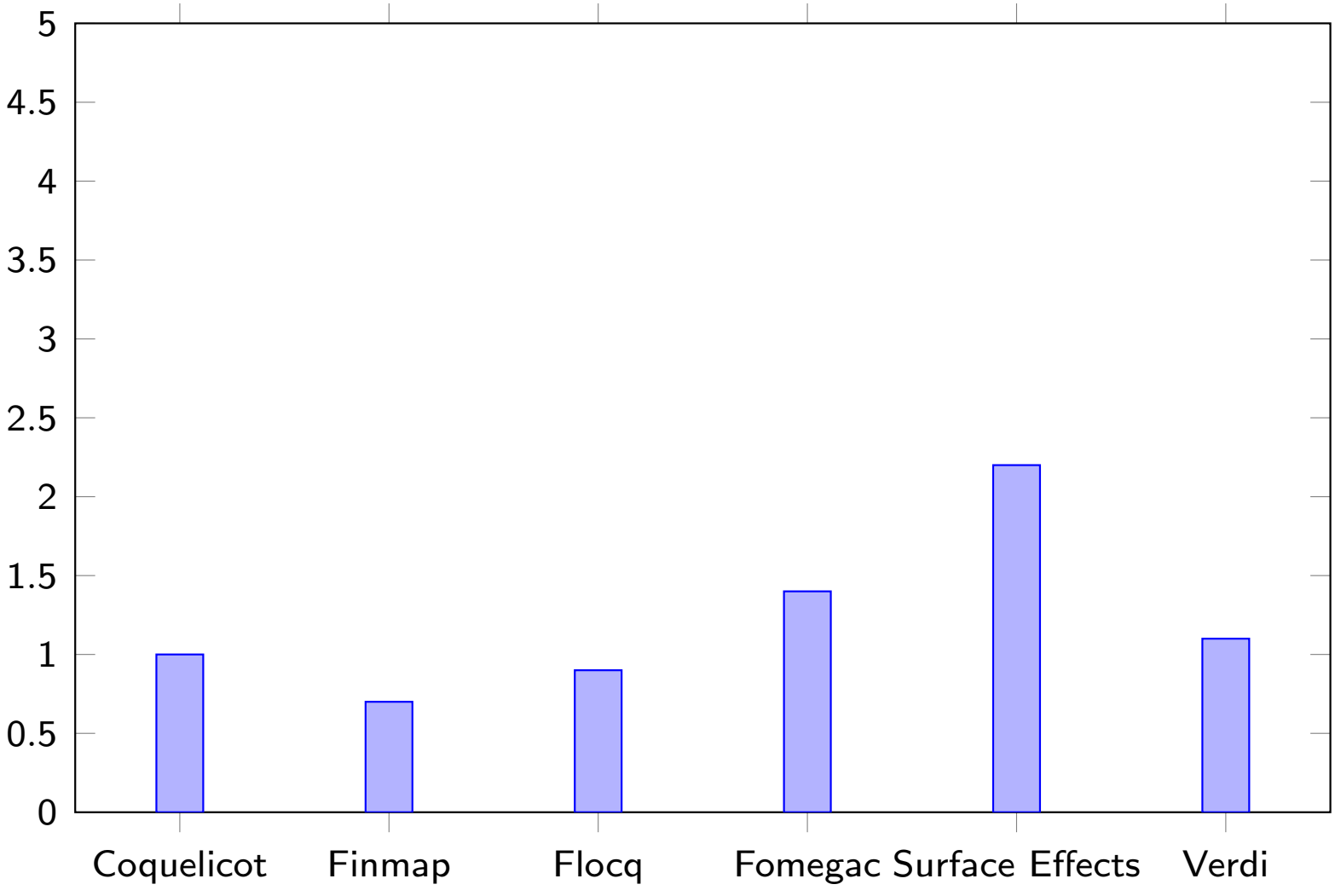
- **Karl Palmskog**, palmskog@utexas.edu
- Ahmet Celik, ahmetcelik@utexas.edu
- Milos Gligoric, gligoric@utexas.edu

Resources:

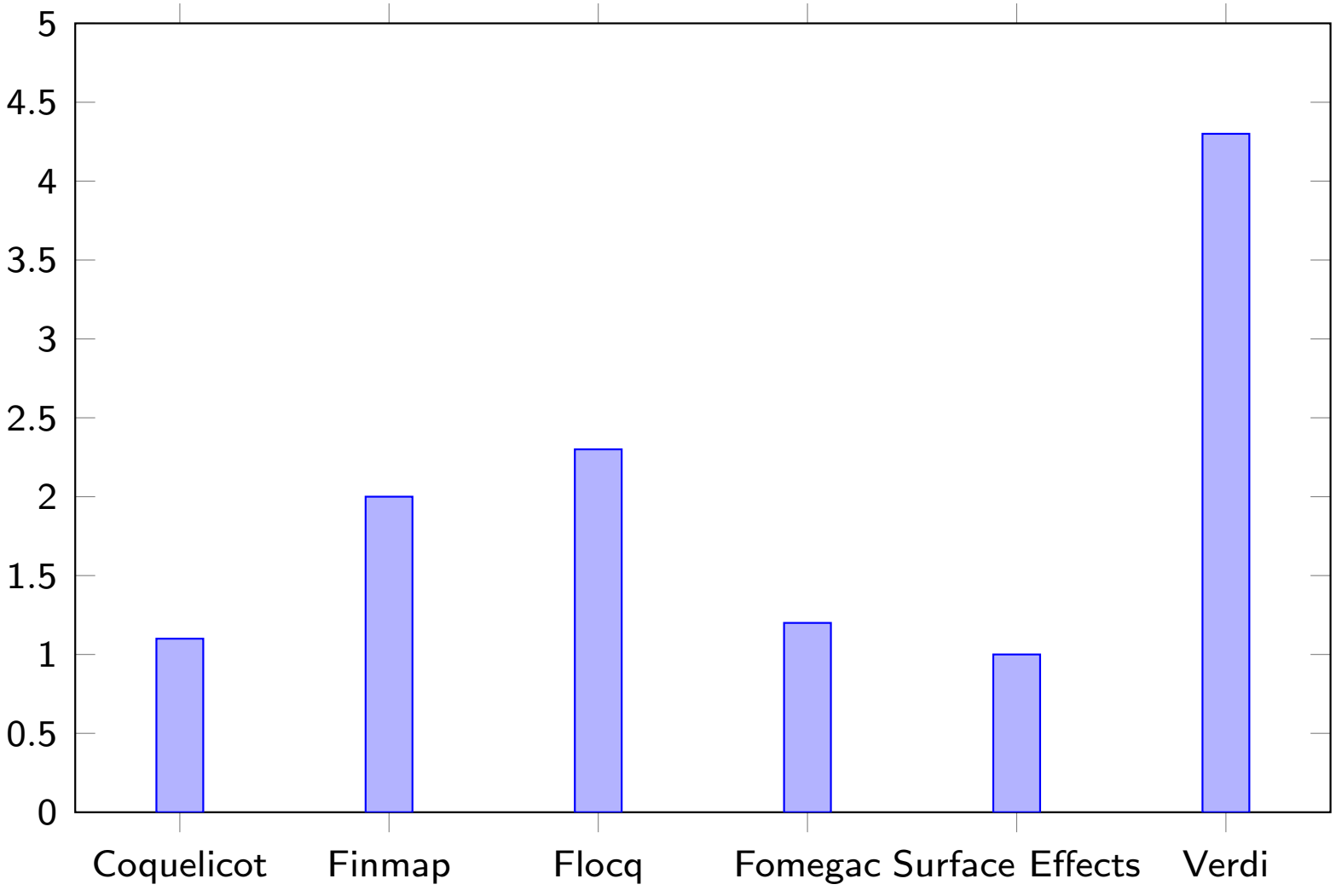
- Website: http://cozy.ece.utexas.edu/icoq/
- GitHub: https://github.com/proofengineering/icoq

This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1566363 and CCF-1652517.

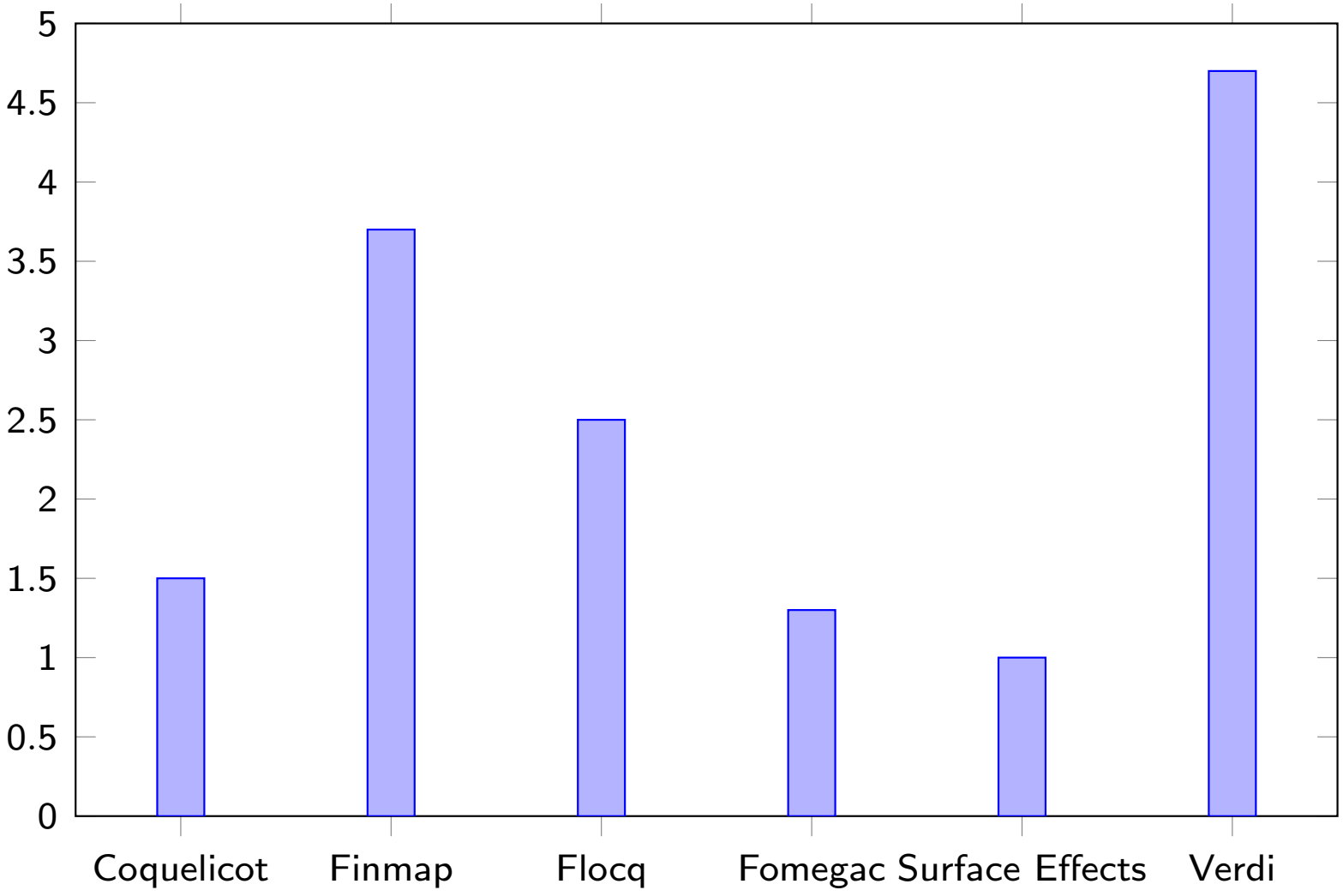
`f·none` vs. `p·none` for 4-way Parallel Checking



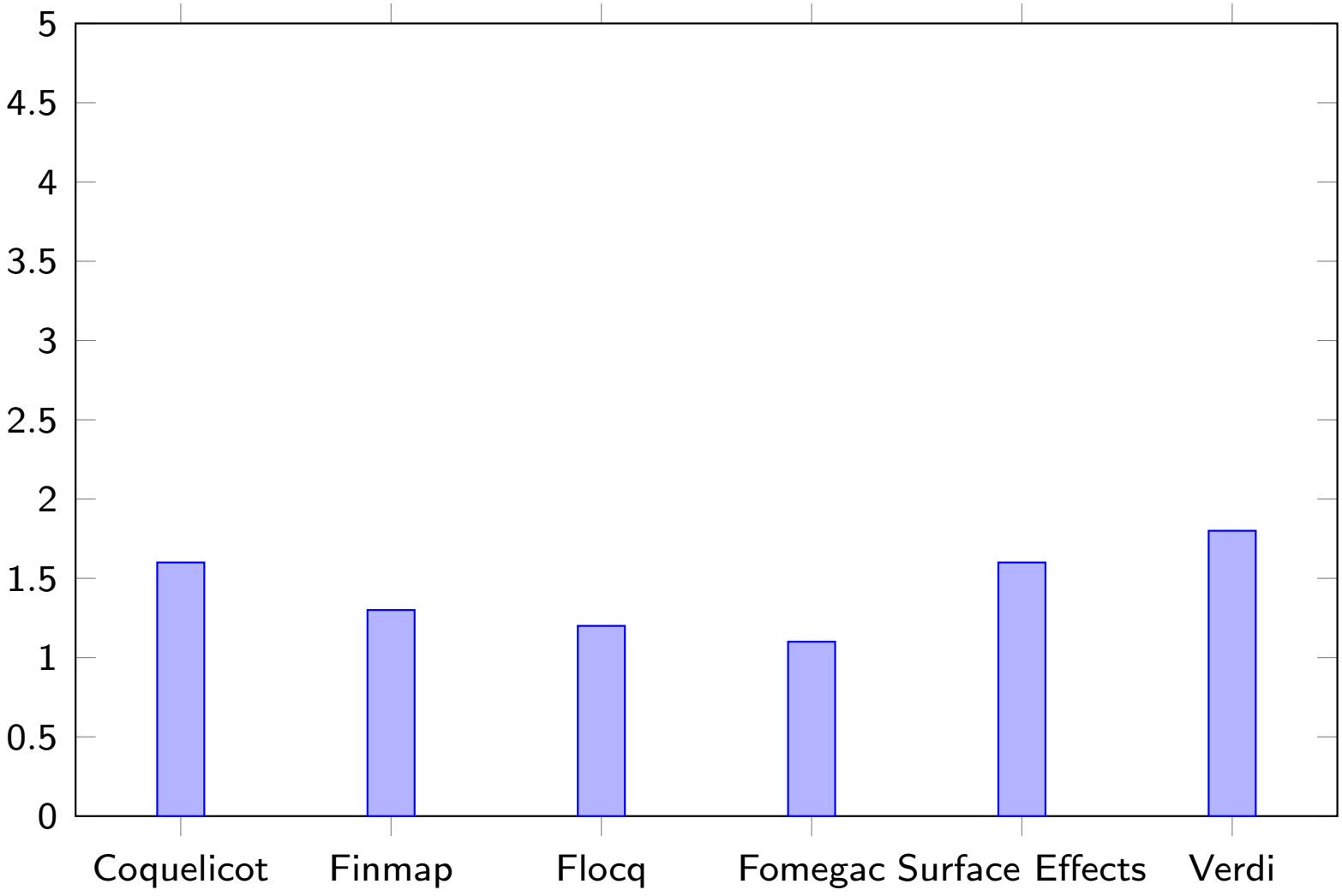
`f•none` vs. `f•file` for 4-way Parallel Checking



`p·none` vs. `p·file` for 4-way Parallel Checking



`p·file` vs. `p·icoq` for 4-way Parallel Checking



Regression Proving in Evolving Projects

Typical **proving** scenario:

1. change definition or lemma statement
2. begin process of re-checking all proofs
3. checking fails much later (for seemingly unrelated proof)

Typical **testing** scenario:

1. change method statements or method signature
2. begin process of re-running all tests
3. testing fails much later (for seemingly unrelated test)

Software Engineering Techniques For Effective Testing

Test selection: run only tests affected by changes

- file/class selection
- method selection
- hybrid

Examples: Ekstazi [ISSTA '15], STARTS [ASE '17], HyRTS [ICSE '18]

Test parallelization: leverage multi-core hardware

- parallel threads
- parallel processes (VM forking)
- hybrid

Examples: Gradle, Maven, JUnit

Regression Proving vs. Regression Testing

- proof checking is deterministic
- proof checking has no side effects (e.g., I/O)
- only file-level deps. relevant for (asynch) proof checking