Université Libre de Bruxelles
Computer Science Department

MEMO-F524 Masters thesis

An Efficient and Parallel Abstract Interpreter in Scala — 3×3 Parallel Implementations —

Olivier Pirson — opi@opimedia.be
orcid.org/0000-0001-6296-9659

March 22, 2019
(Last modifications: March 26, 2019)

https://speakerdeck.com/opimedia/efficient-parallel-abstract-interpreter-in-scala-3x3-implementations

Vrije Universiteit Brussel

Promotors Coen De Roover
Wolfgang De Meuter
Advisor Quentin Stievenart

From undecidable problem to fast decidable technique

We want prove **properties about programs**.
But it is an **undecidable** problem.

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

↓

Abstract interpretation is a static analysis technique that evaluates abstraction of programs.
Decidable but **high time complexity**.

↓

A *possible* way to make that **faster** is to **parallelize**...

Parallelization of Scala-AM

Scala-AM

"(Abstract) Abstract Machine Experiments using Scala"

(forked around September 14, 2017)

↓

Restriction to the simplest abstract machine (AAM), without optimization

Restriction to analyze only Scheme programs.

↓

3×3 parallel implementations (ParAAM∗) with actors (Akka)

↓

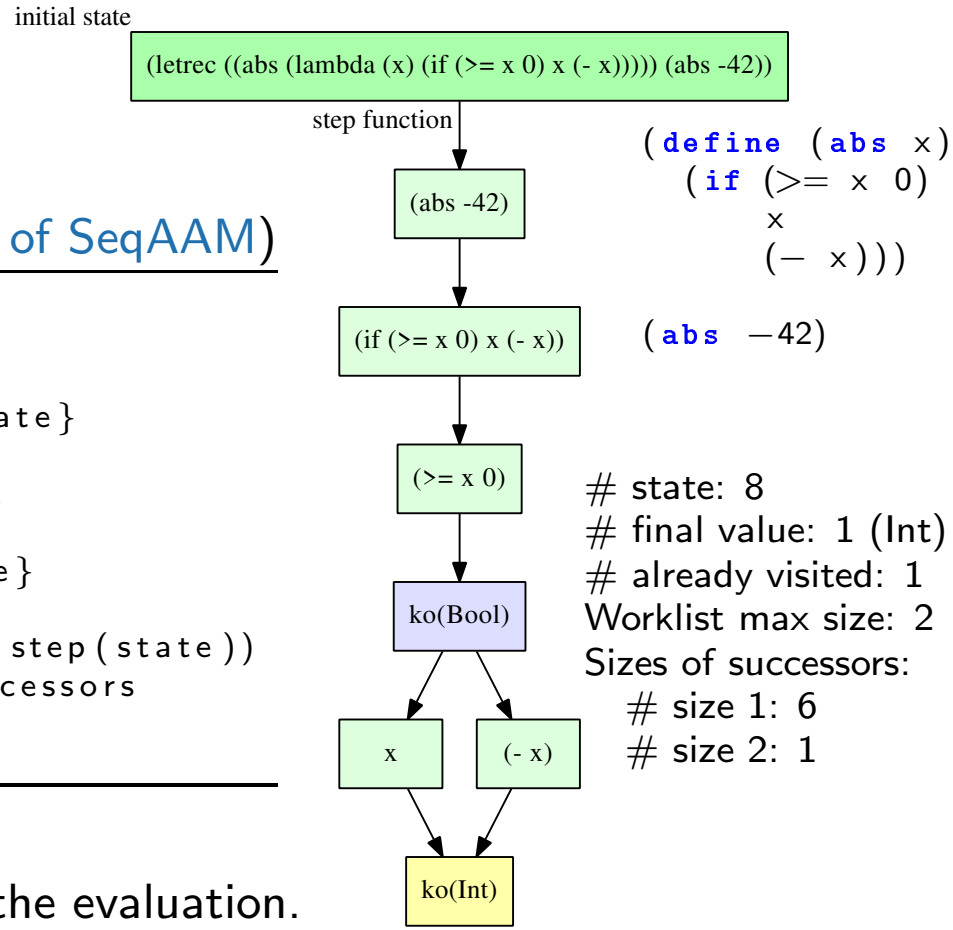
Scala-Par-AM

https://bitbucket.org/OPiMedia/scala-par-am/

Worklist and step function, walk over the state graph

Essential parts of abstract interpretation for this presentation:

- main loop on a **worklist** of states
- the **step function**:
  takes a state
  and returns a set of states

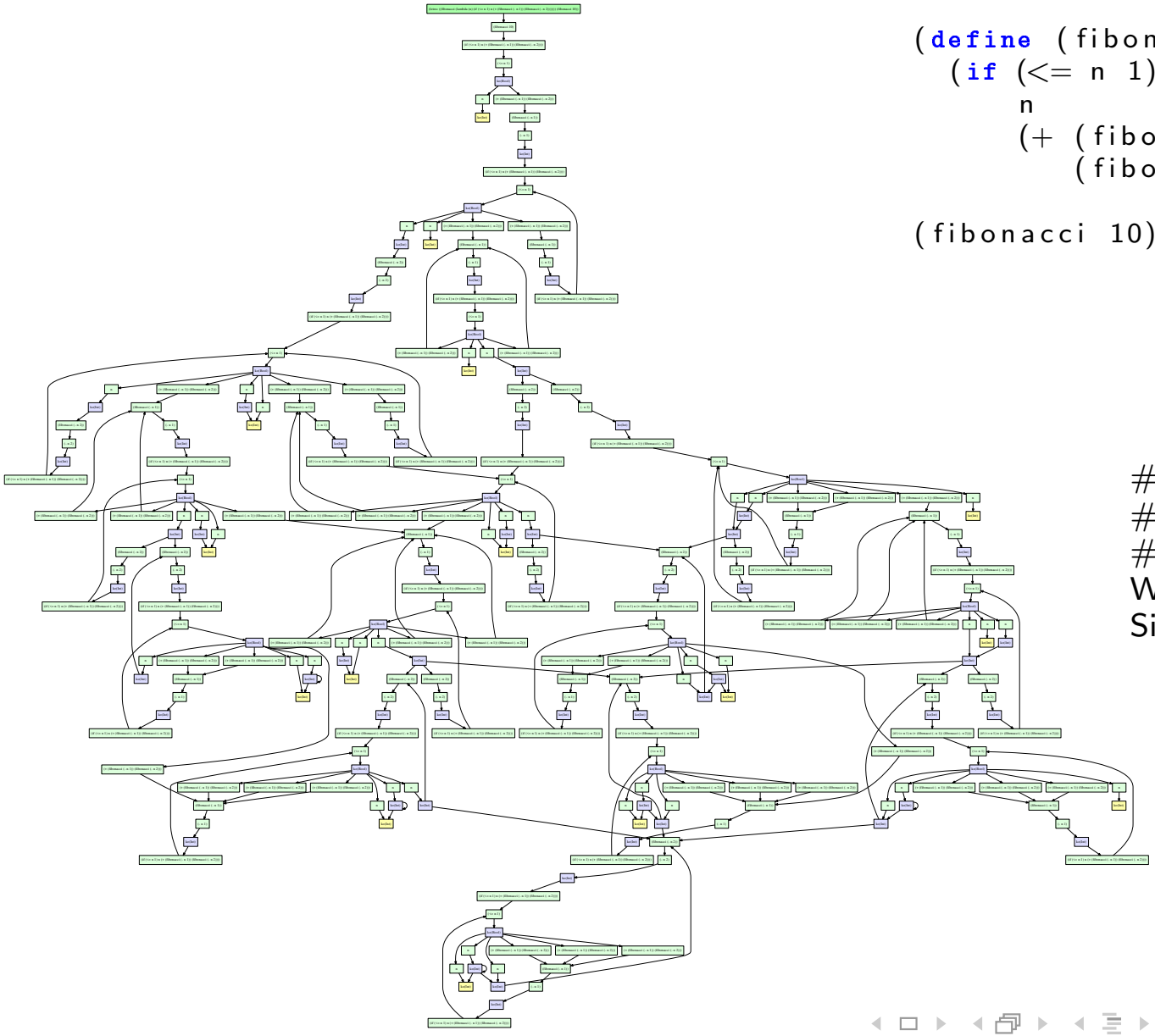


Sequential algorithm (Scala code of SeqAAM)

```
worklist = empty list
visited = empty set

worklist = worklist ∪ {initial state}
repeat
    state = pick one from worklist
    if state not in visited then
        visited = visited ∪ {state}
        evaluation of state
            (essentially successors = step(state))
        worklist = worklist ∪ successors
        other update (final values...)
until worklist is empty
```

The completed state graph
does *not* depend of the order of the evaluation.
Crucial for parallelization.

Very simple Scheme example, bigger state graph



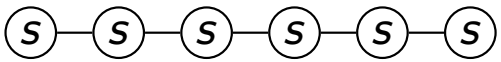
```
(define (fibonacci n)
  (if (<= n 1)
      n
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2)))))

(fibonacci 10)
```

state: 276
final value: 1 (Int)
already visited: 71
Worklist max size: 24
Sizes of successors:
  # size 1: 231
  # size 2: 13
  # size 3: 3
  # size 4: 2
  # size 6: 12

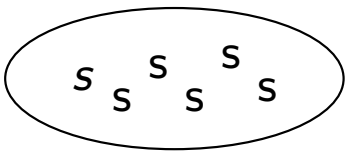
Worklist implementations

Possible (immutable) data structures to implement the worklist
(tested on Sergey/jfp/primtest.scm)

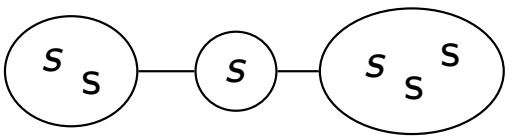
- **list** of states — 2.304s (Scala code of SeqAAM)



- **set** of states — 2.466s (Scala code of SeqAAMS)



- **list of sets** of states — 2.315s (Scala code of SeqAAMLS)

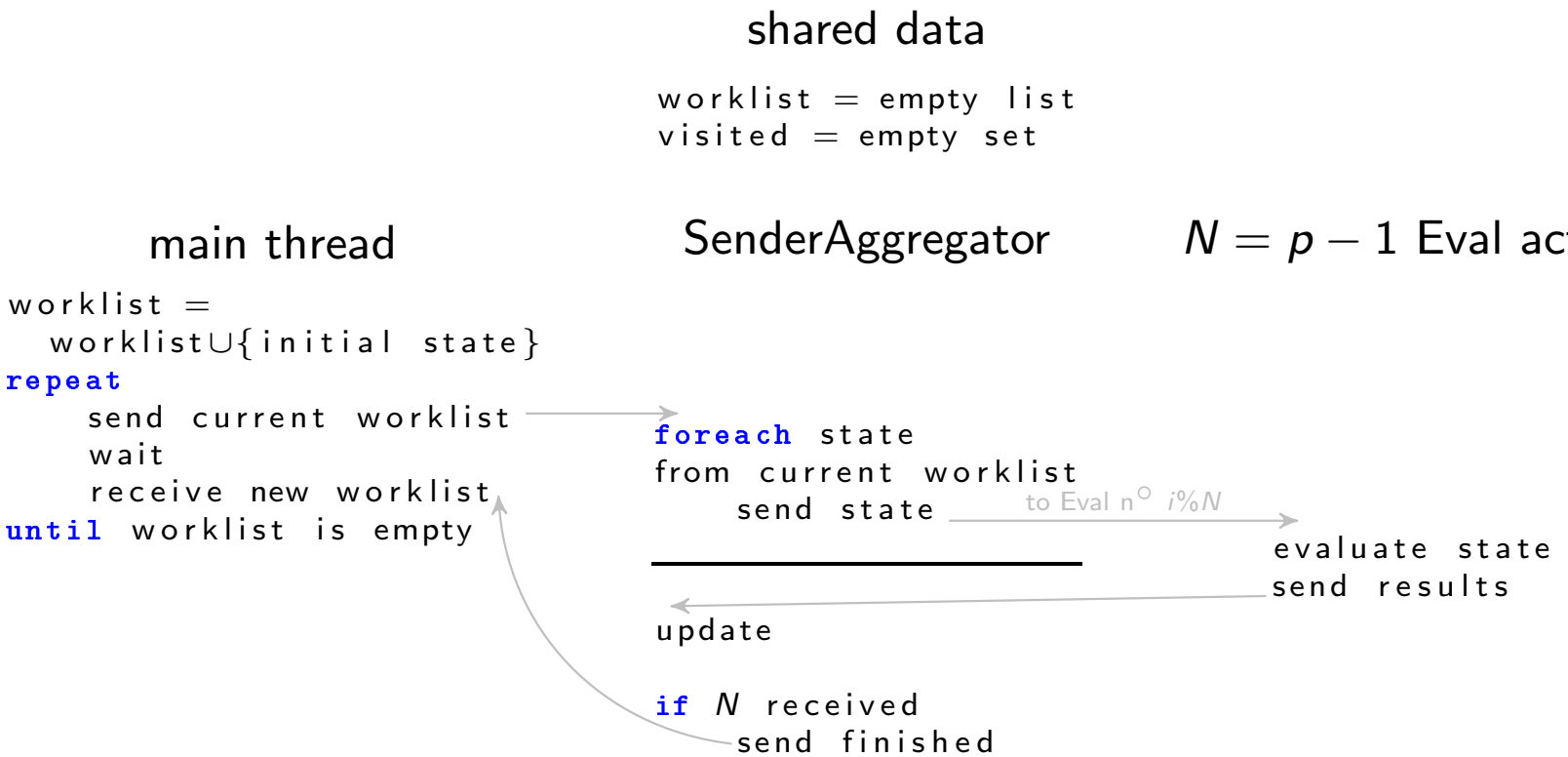


avoid useless concatenations

Each set corresponds to a result of the step function.
Selected for parallel implementations
and **SeqAAMLS becomes the reference for comparisons**.

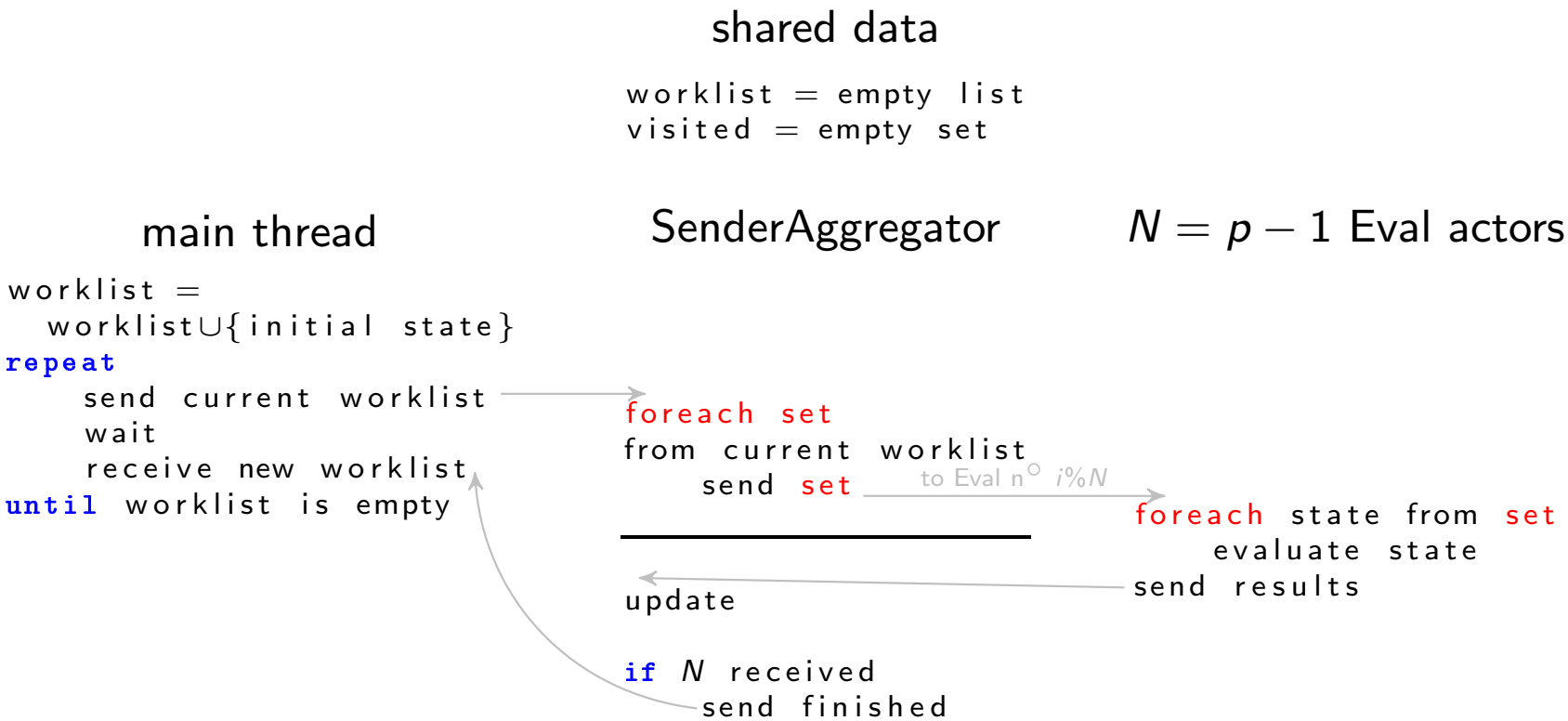
These time results and following are **averages** on 10 repetitions (after 3 repetitions skipped), made on **Intel Xeon Gold 6148 2.40GHz**, **20 cores** available (VUB Hydra cluster), with Scala 2.12.7 – Akka 2.5.21 – Java 1.8.0 – GraalVM 1.0.0-rc14.

First parallel implementation: ParAAM-L-SA-state



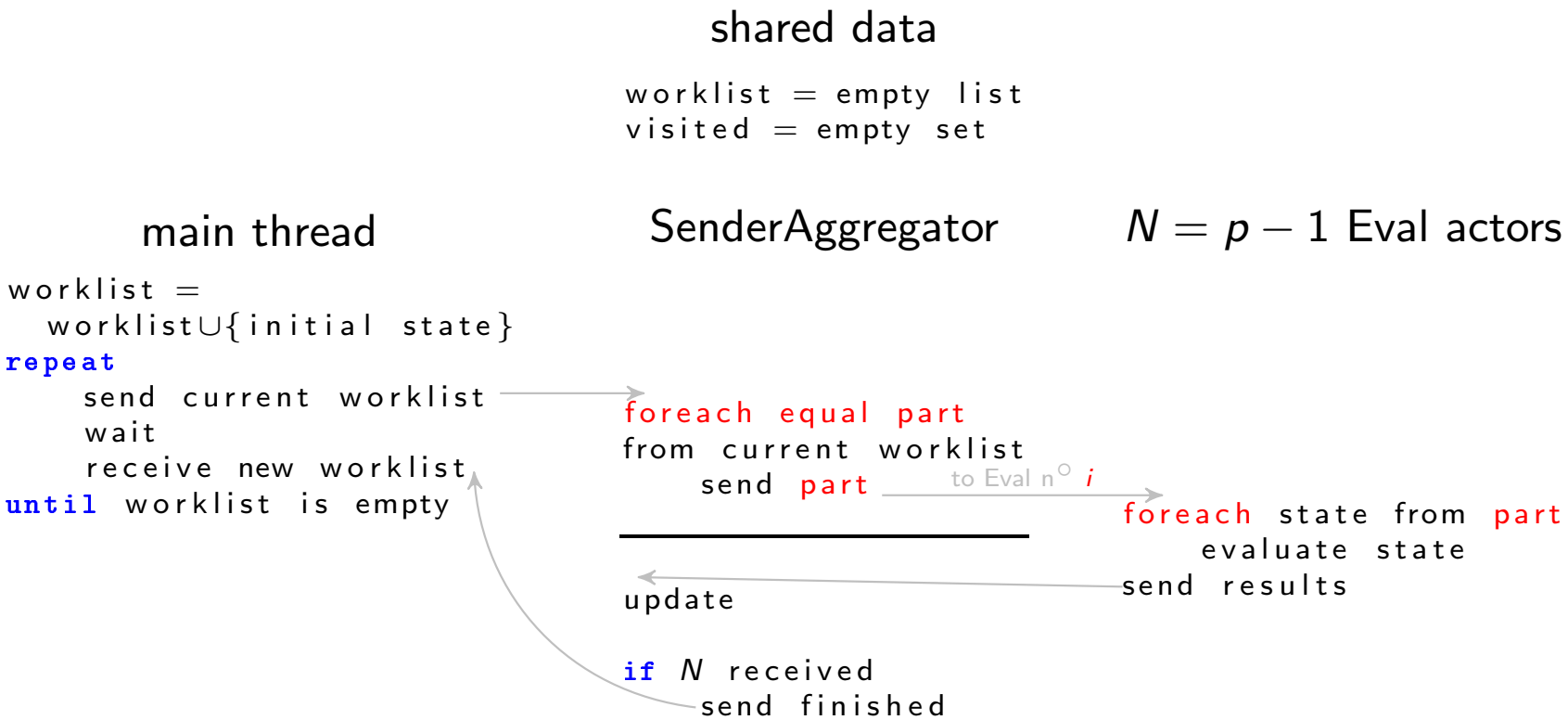
I name this implementation ParAAM-L-SA-state, for ParAAM – Loop – SenderAggregator – state.

First alternative: ParAAM-L-SA-set



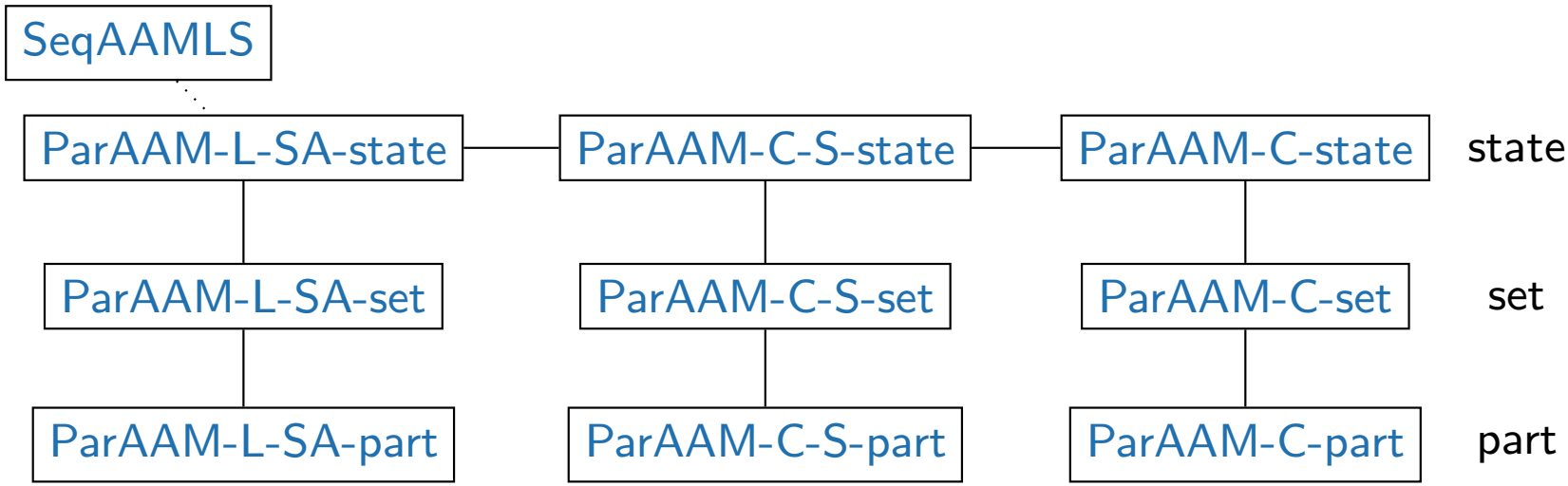
First improvement: send more than one state at a time. Set a set of states. ParAAM-L-SA-set, for ParAAM – Loop – SenderAggregator – set.

Second alternative: ParAAM-L-SA-part



Other possibility: send "equals" part of the worklist.
ParAAM-L-SA-part, for ParAAM – Loop – SenderAggregator – part.

3×3 parallel implementations



L-SA
Loop – SenderAggregator

1 special actor
p-1 eval actors
main loop (with a barrier)
SA send worklist
each eval actor send results to SA

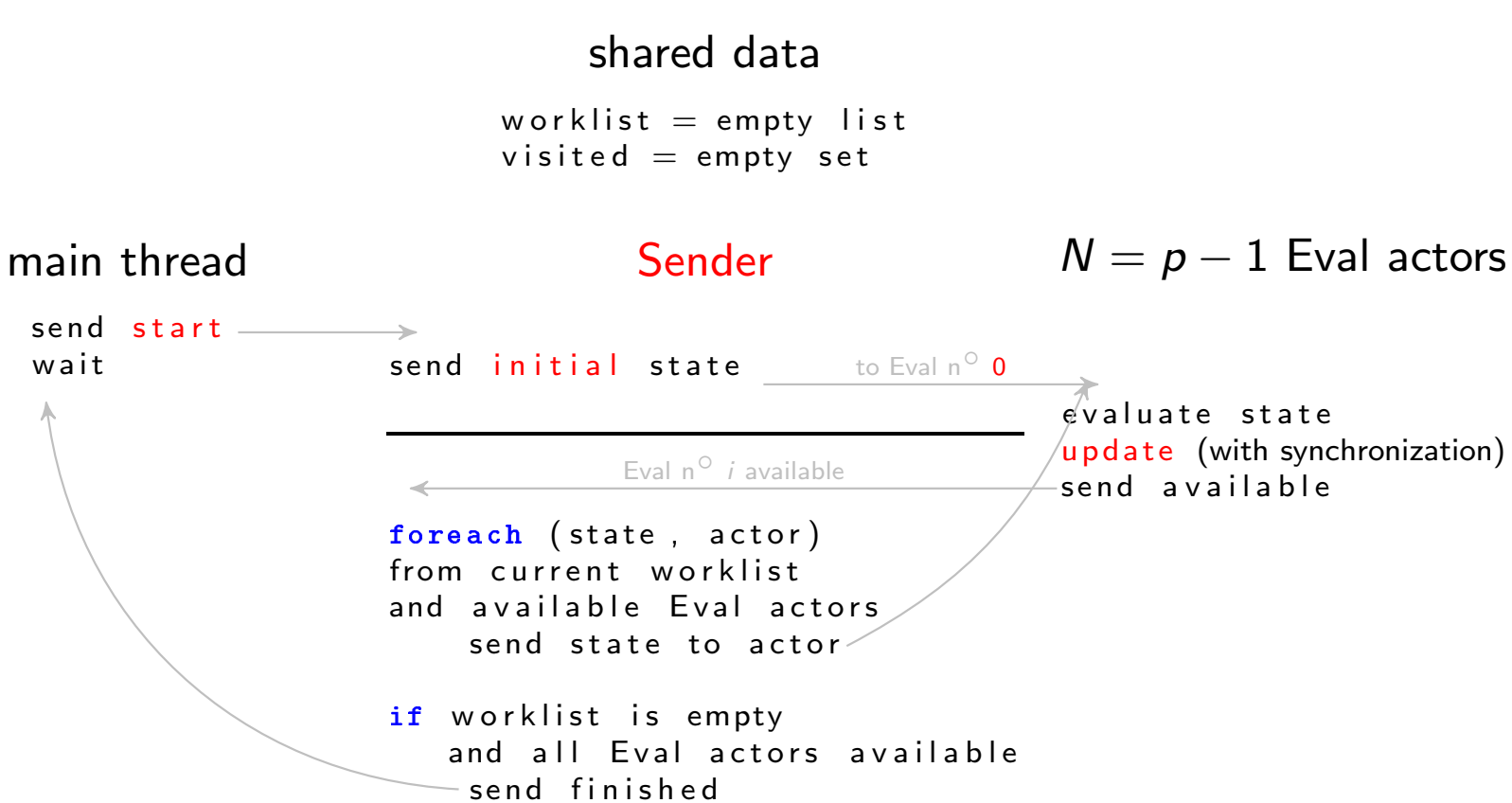
C-S
Concurrent – Sender

1 special actor
p-1 eval actors
no barrier
S send worklist
each eval actor update

C
Concurrent

0 special actor
p eval actors
no barrier
each eval actor send worklist
each eval actor update

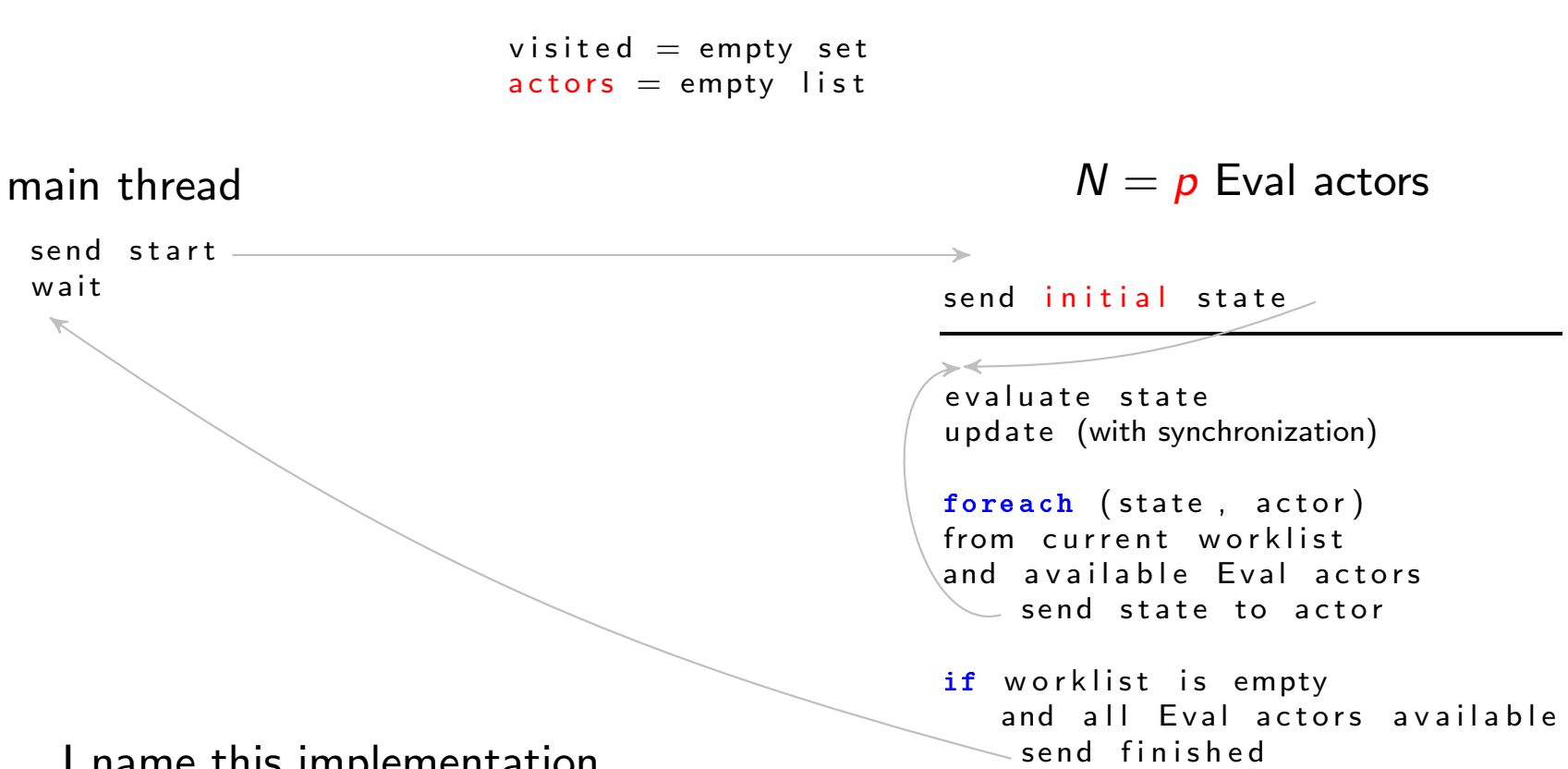
Other strategy: ParAAM-C-S-state



I name this implementation ParAAM-C-S-state, for ParAAM – Concurrent – Sender – state.

And these similar improvements:
ParAAM-C-S-set, for ParAAM – Concurrent – Sender – set and
ParAAM-C-S-part, for ParAAM – Concurrent – Sender – part.

Last strategy: ParAAM-C-state



I name this implementation
ParAAM-C-state, for ParAAM – Concurrent – state.

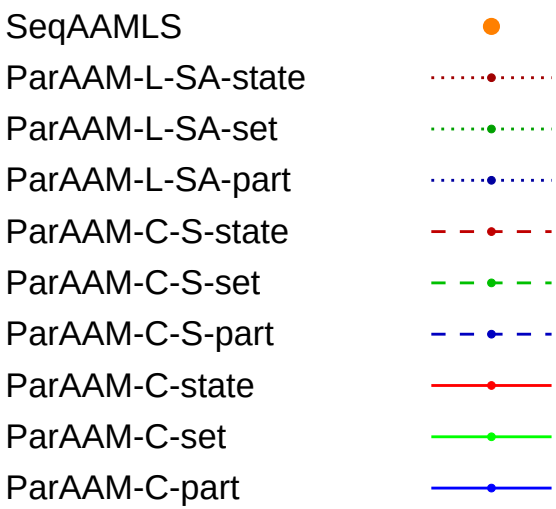
And these similar improvements:
ParAAM-C-set, for ParAAM – Concurrent – set and
ParAAM-C-part, for ParAAM – Concurrent – part.

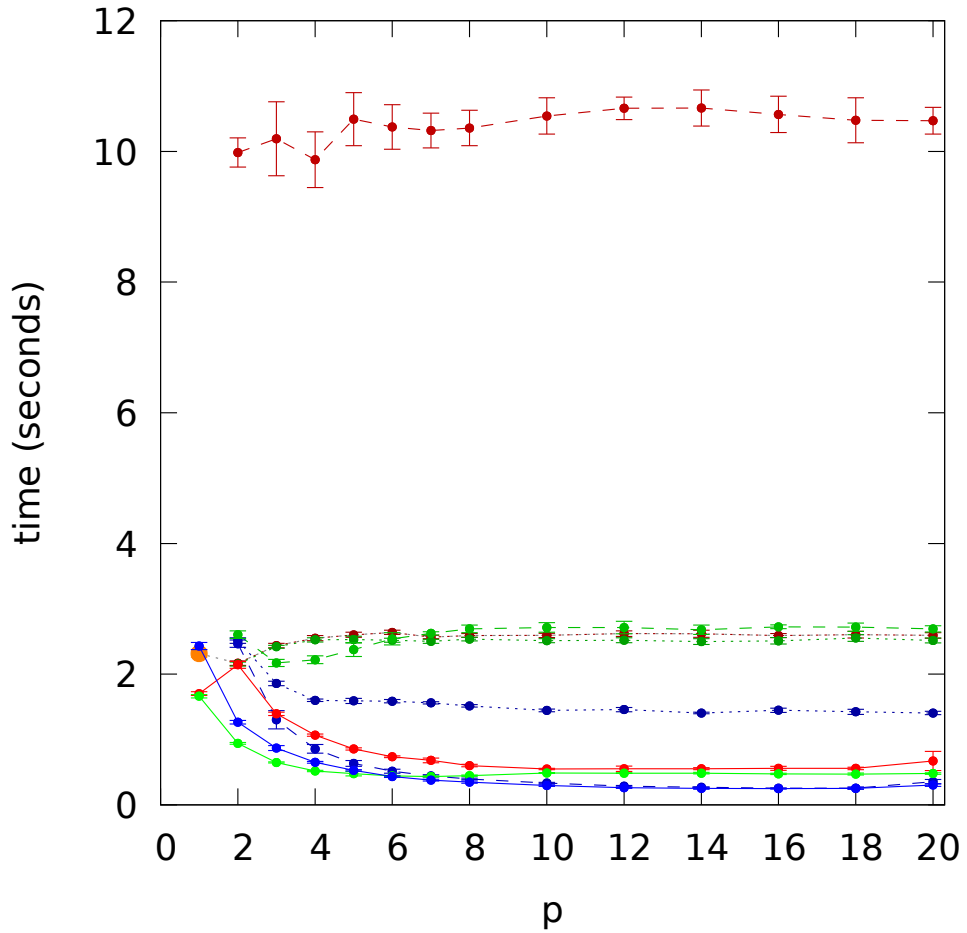
Computation time of all implementations

Results computed on Sergey/jfp/primtest.scm

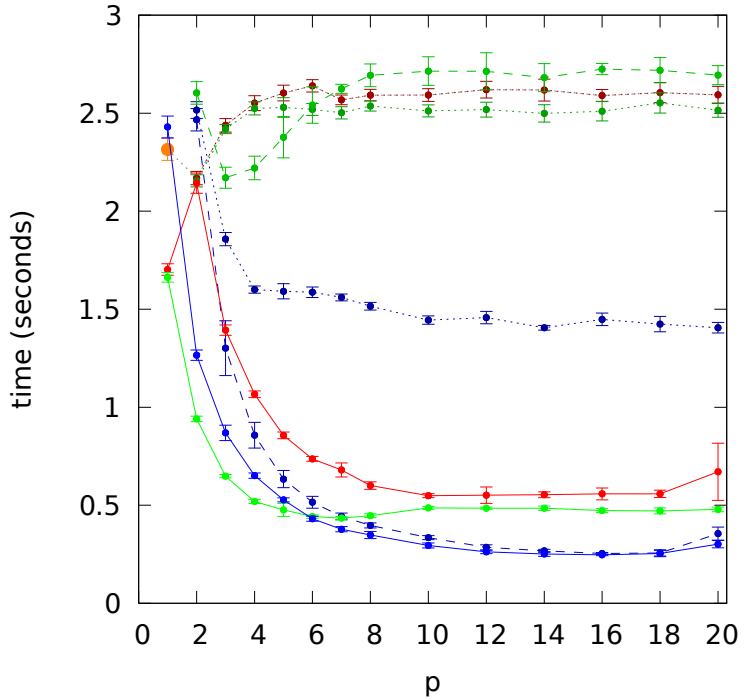
Statistic computed with SeqAAMLS

state: 243472
final: 1 (Int)
error: 1
already visited: 48430
Worklist max size: 5646
Max sizes of successors: 6



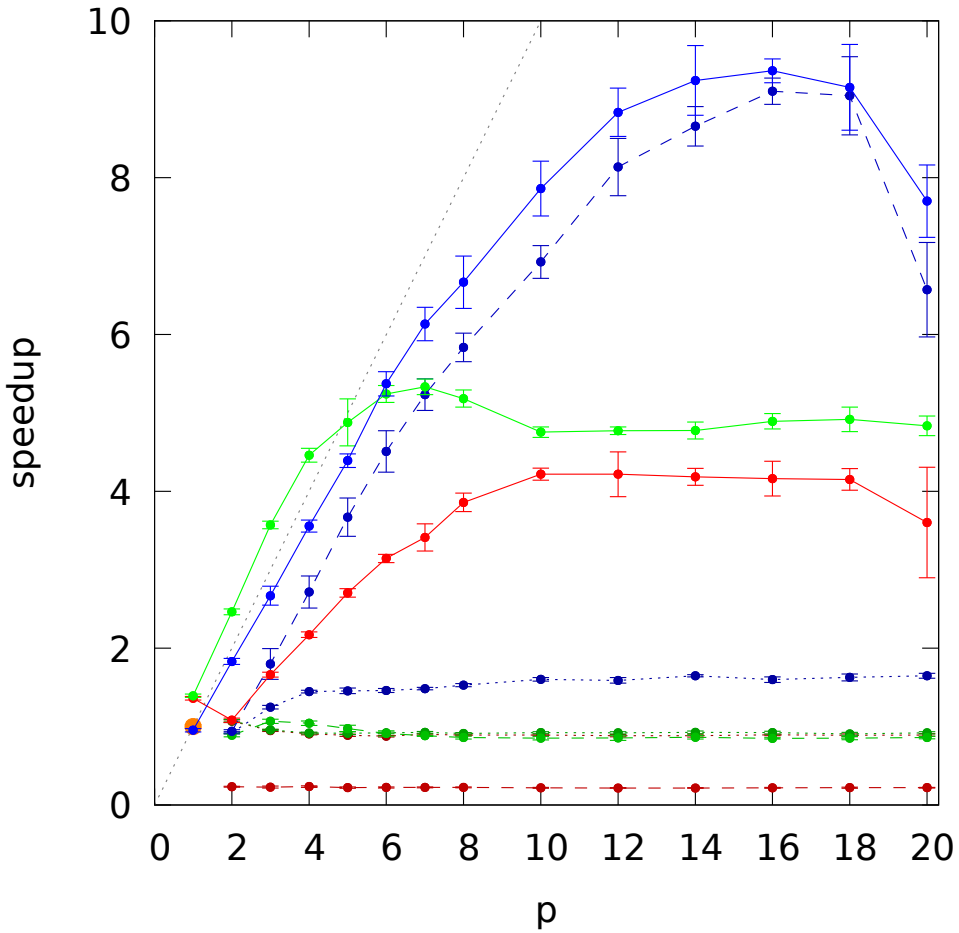


Speedup



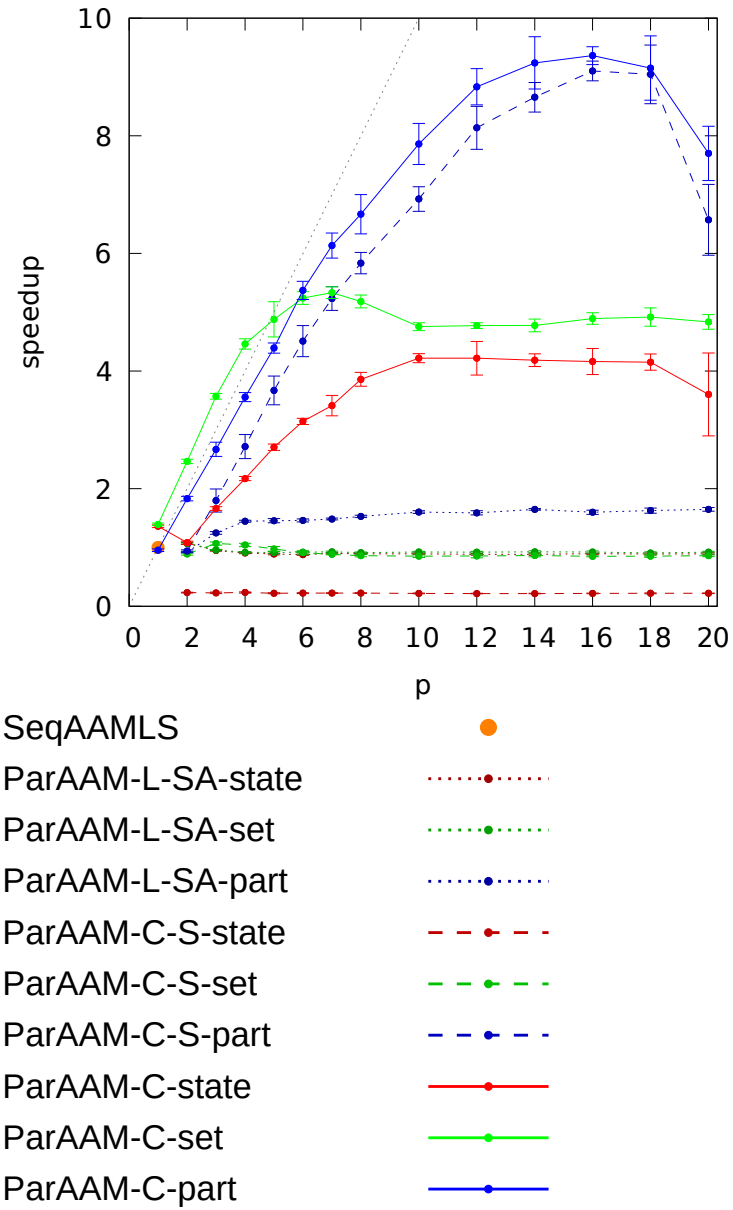
SeqAAMLS
ParAAM-L-SA-state
ParAAM-L-SA-set
ParAAM-L-SA-part
ParAAM-C-S-state
ParAAM-C-S-set
ParAAM-C-S-part
ParAAM-C-state
ParAAM-C-set
ParAAM-C-part

$$\text{speedup} = \frac{\text{time of SeqAAMLS}}{\text{time}}$$

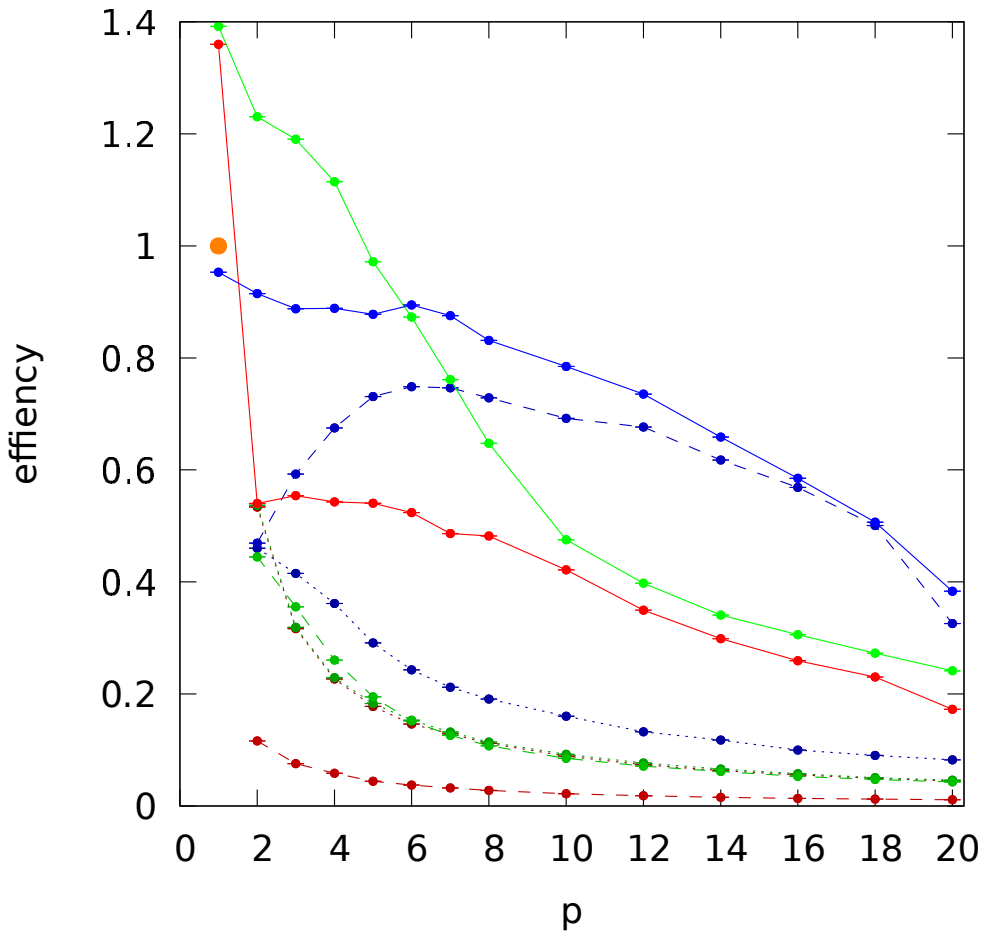


(Super-linear speedup is unexplained for now!)

Efficiency



$$\text{efficiency} = \frac{\text{speedup}}{p}$$



Done / Todo

Implementation is (almost) done ☺
(Need little checkings,
and maybe change the update of visited set for the worst implementations.)

Todo

- Select which Scheme examples to use to produce final benchmarks and estimate computation times
- Run computations and collect data (benchmarks and some statistics)
- Sort data and finalize analyzes
- Write and write again ✍
  (some parts of theoretical background
  can be taken from preparatory work)

Thank you!

Questions time. . .

- Implementation:
   Scala-Par-AM – https://**bitbucket.org/OPiMedia**/scala-par-am/
- This presentation on SD
  https://**speakerdeck.com/opimedia**/
  efficient-parallel-abstract-interpreter-in-scala-3x3-implementations
- Document, LaTeX sources, other references and previous presentations on
  https://bitbucket.org/OPiMedia/efficient-parallel-abstract-interpreter-in-scala