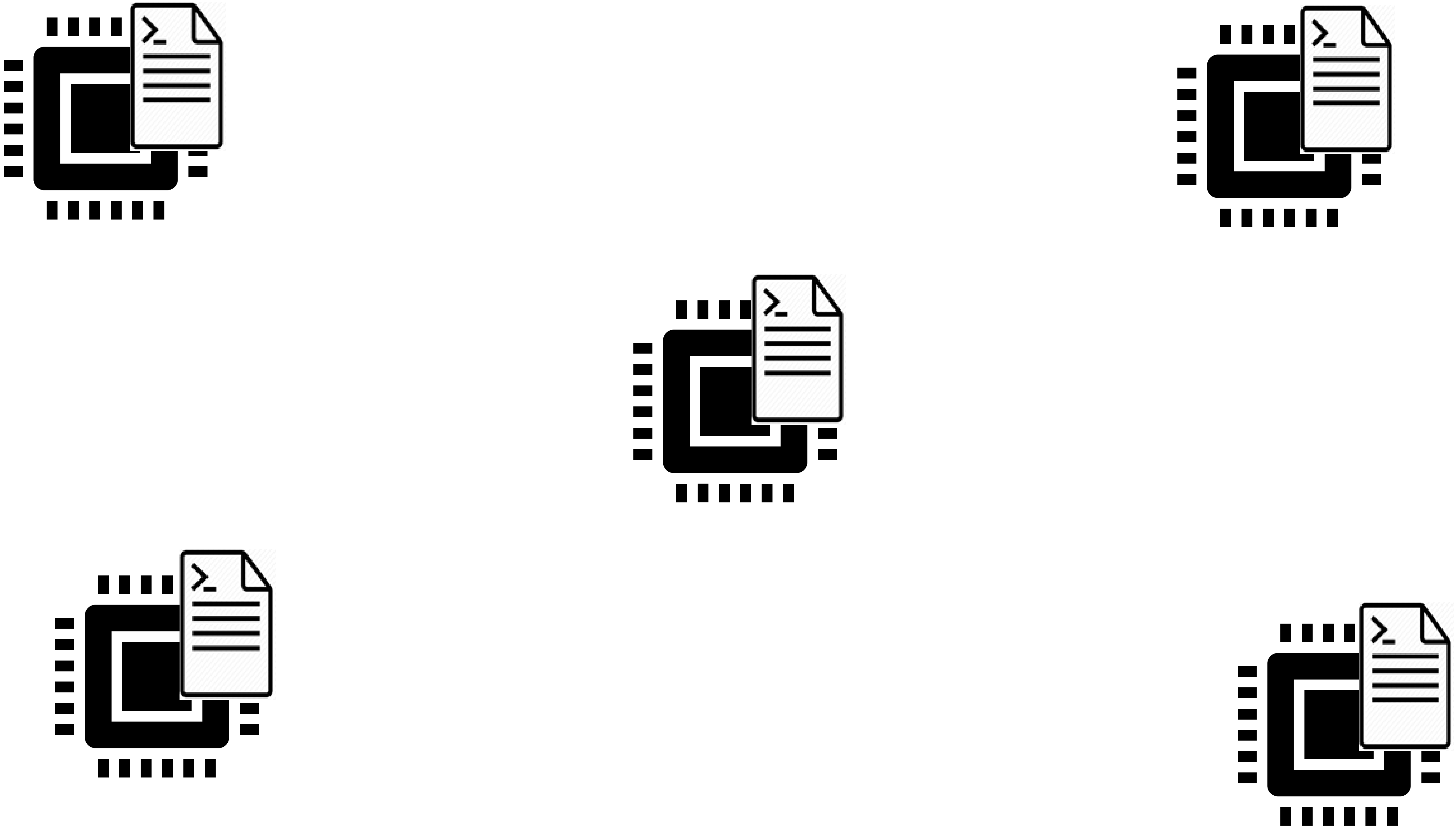
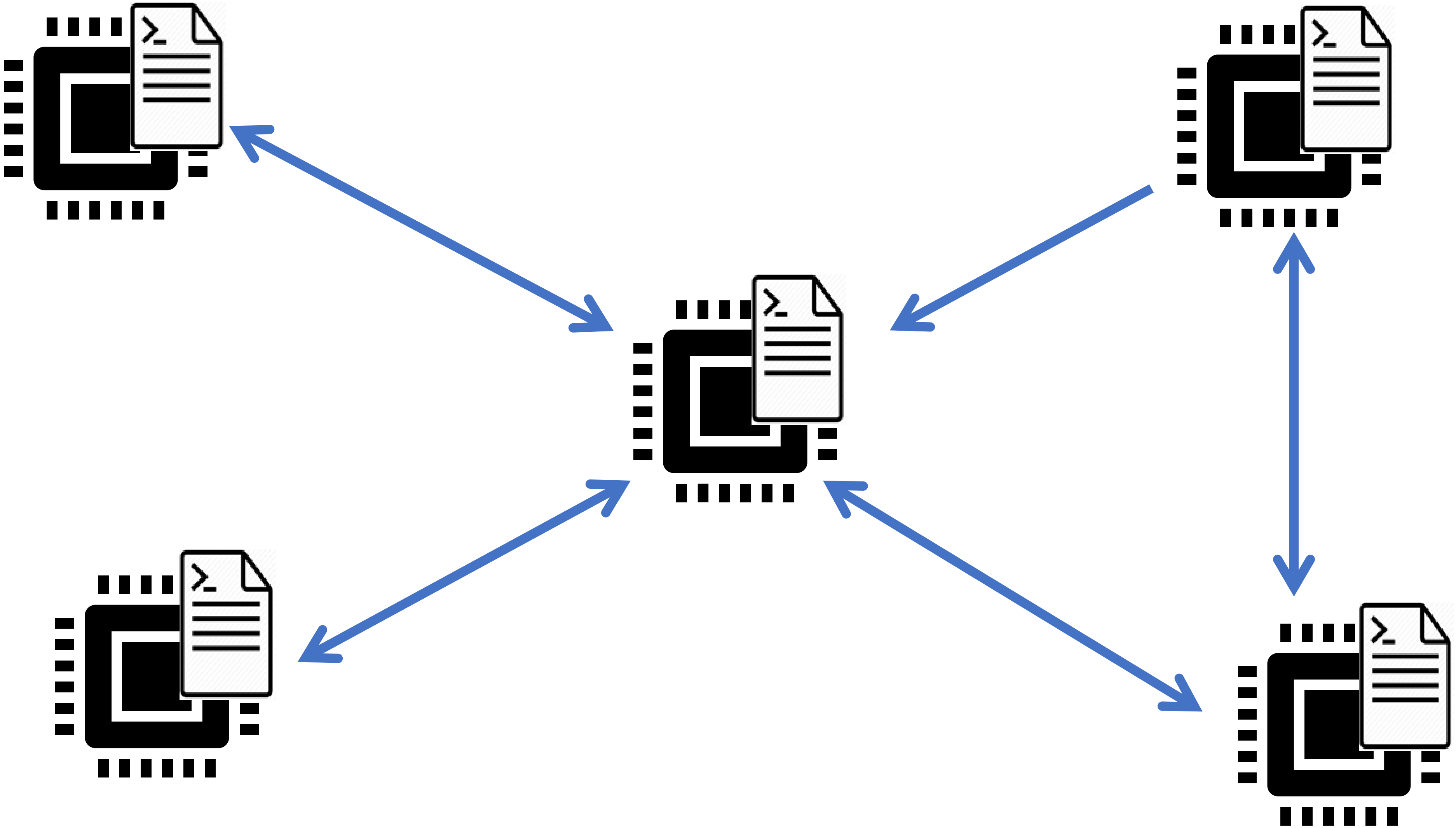
Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization

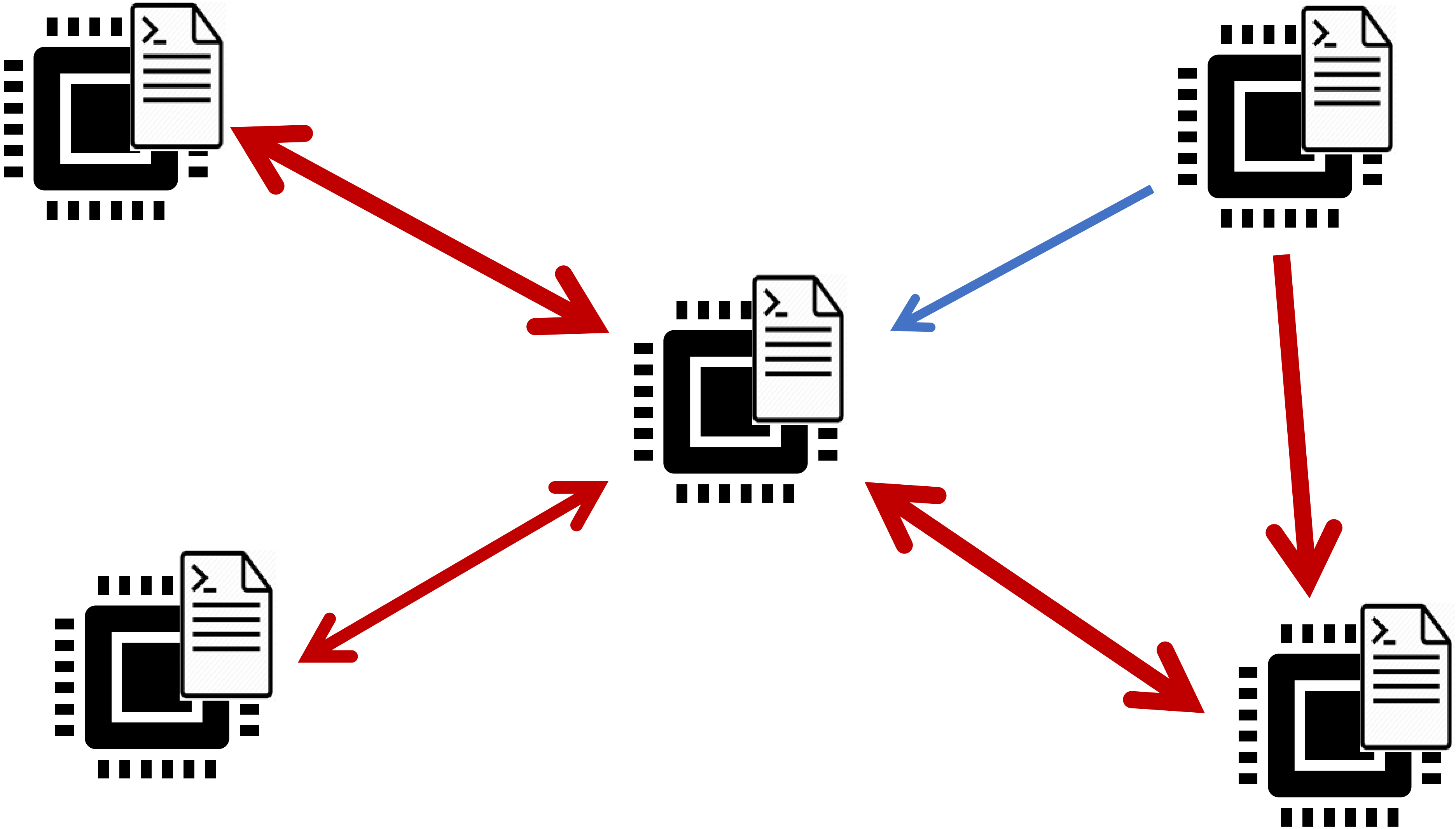
Vimuth Fernando, Keyur Joshi, Sasa Misailovic

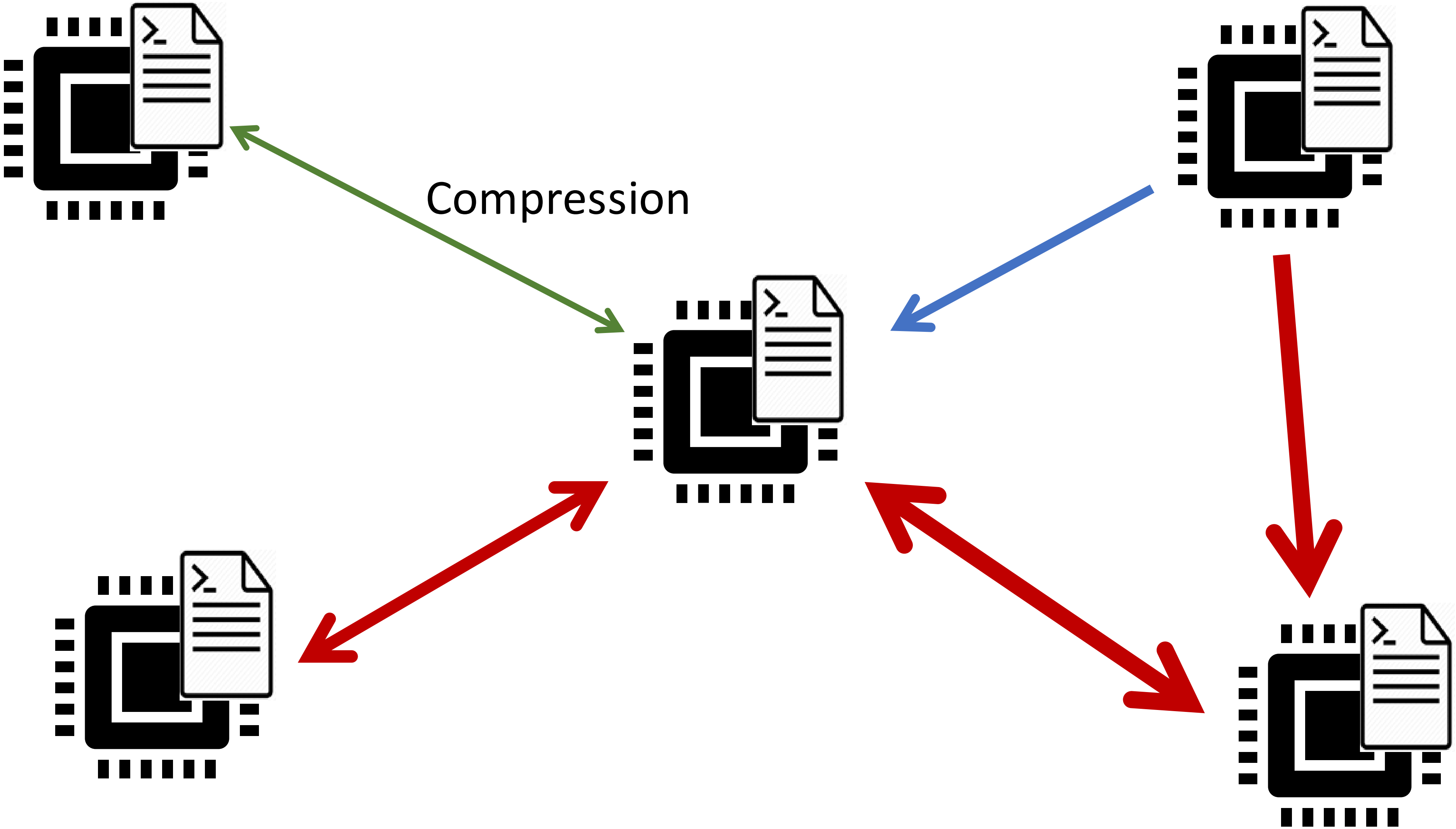
University of Illinois at Urbana-Champaign

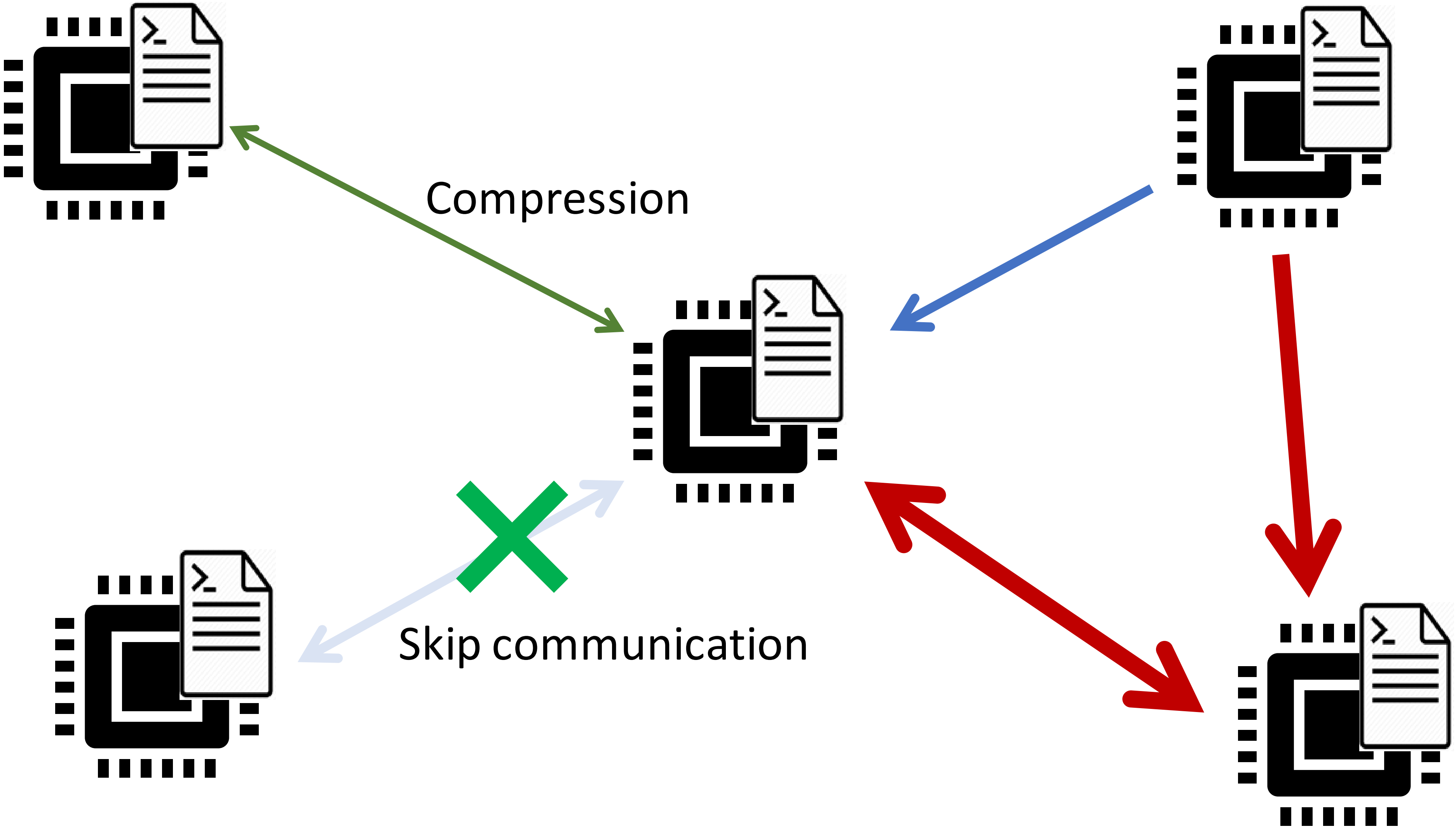
CCF-1629431
CCF-1703637
CCF-1846354





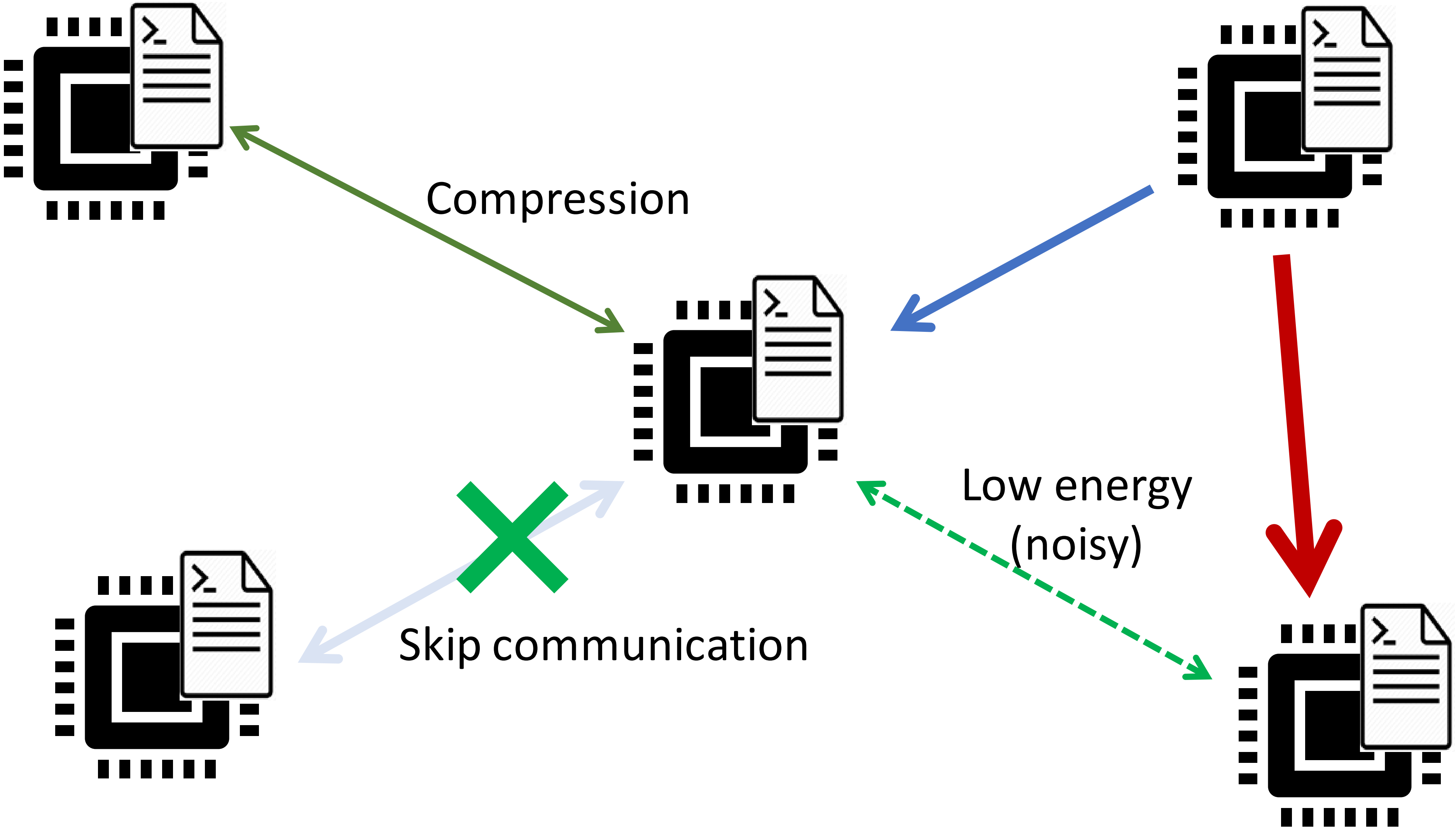



Compression



Compression

Skip communication


Compression
Low energy
(noisy)
Skip communication

How safe is the program?

Approximate program should not crash, get stuck, or produce unacceptable results

How accurate are the results?

Approximate program should produce results with acceptable accuracy/ reliability

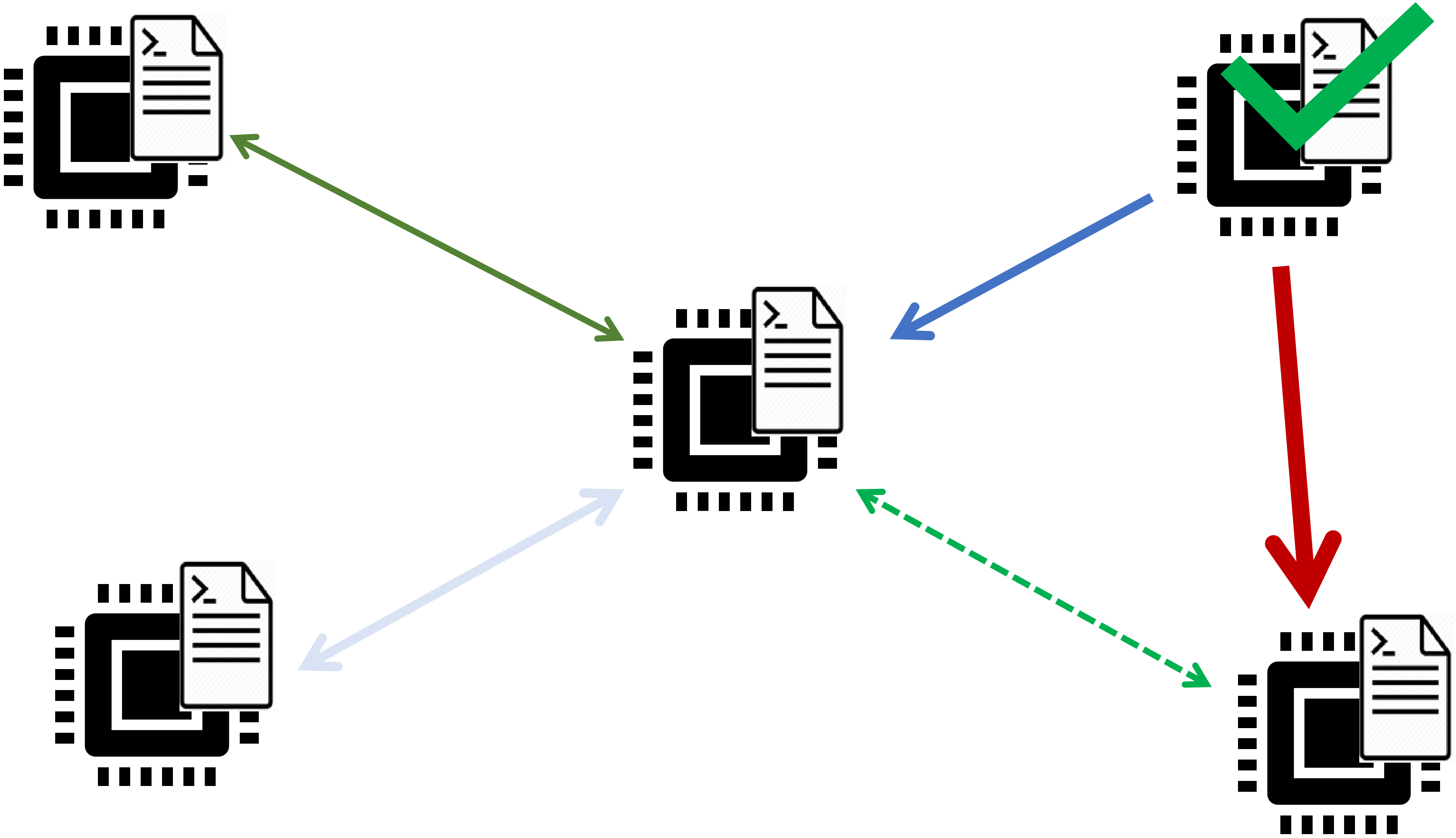
Safety after approximations

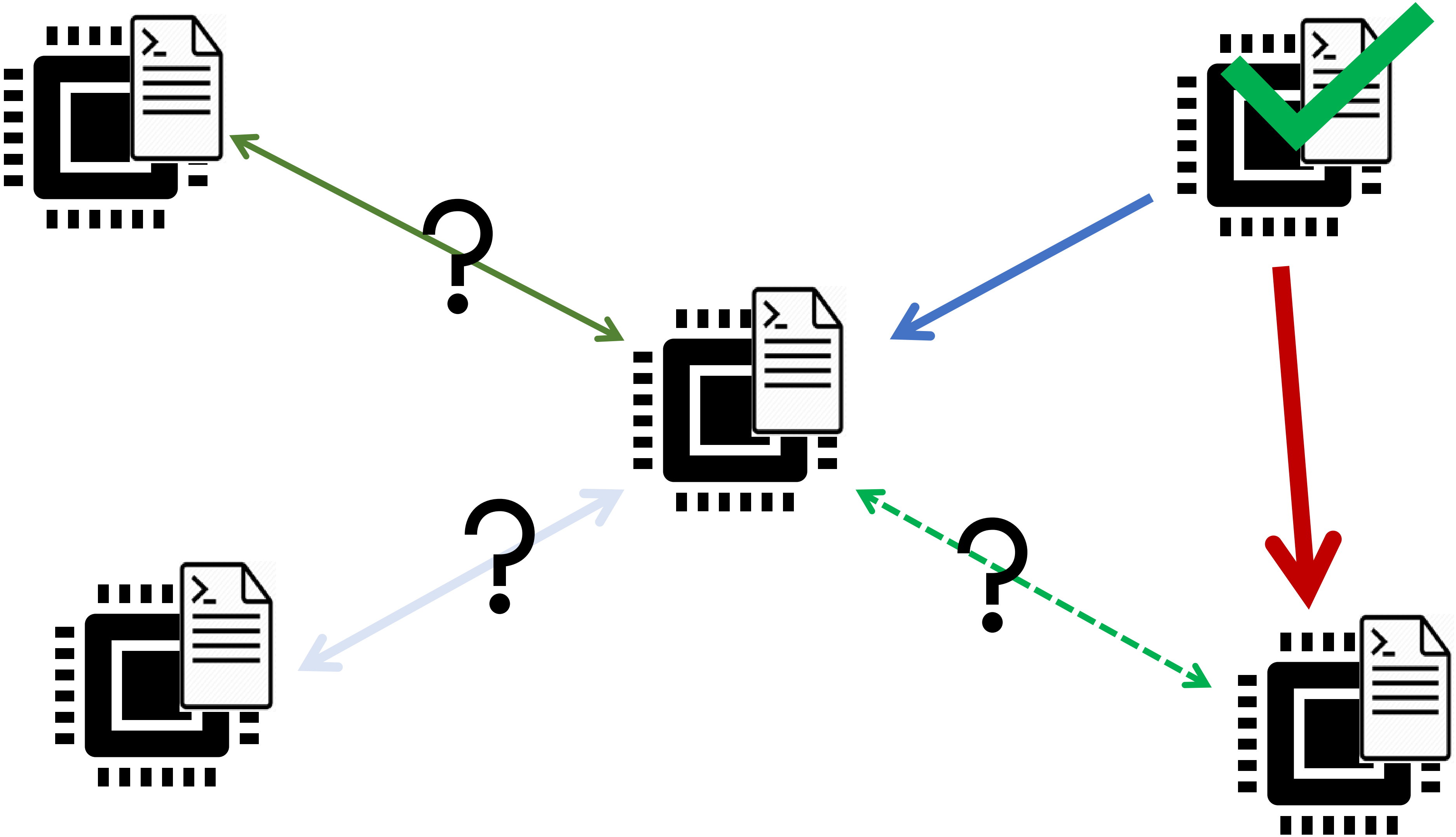
- Types – Non-interfe roximate and precise data [S]
- Relativ oning about ori oximate programs]

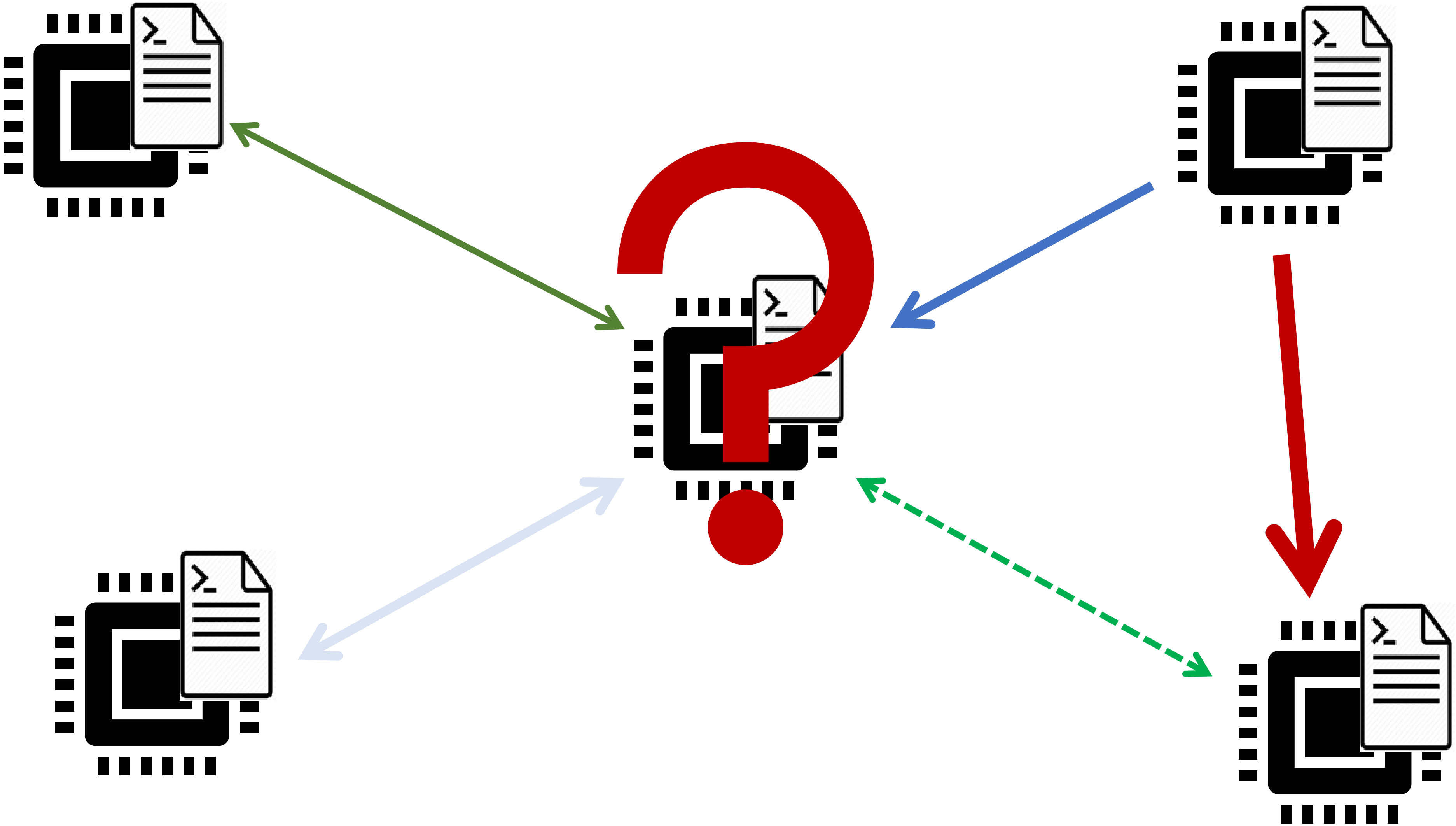
Accuracy aft

- ability - probability of getting the correct result [Carbin et al. 2013]
- Accuracy – combines reliability with distance from correct result [Misailovic et al. 2014]

Only works for sequential programs





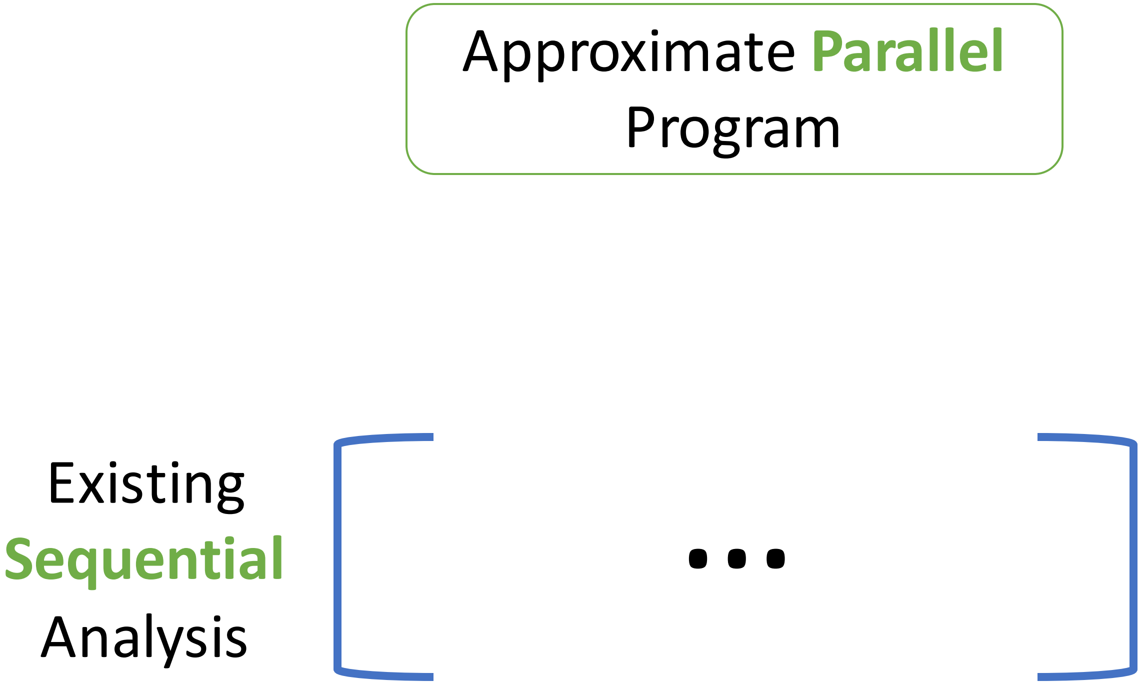


How do we proceed?

- Completely new versions of all analyses?

- Types – Non-interference of approximate and precise data [Sampson et al. 2011]
- Relative safety - Transfer reasoning about original program to approximate programs [Carbin et al. 2012]

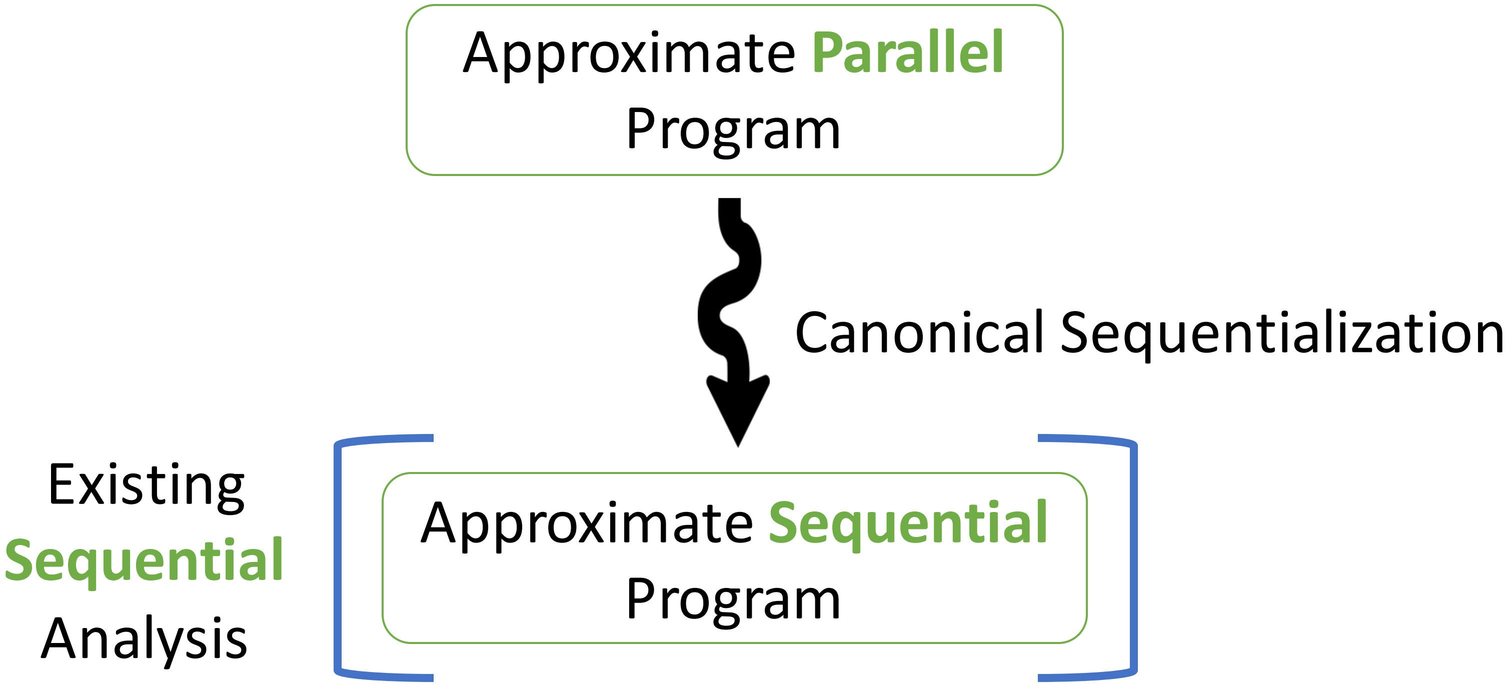
- Reliability - probability of getting the correct result [Carbin et al. 2013]
- Accuracy – combines reliability with distance from correct result [Misailovic et al. 2014]



Approximate **Parallel** Program

Existing **Sequential** Analysis

…



Approximate **Parallel** Program

Canonical Sequentialization

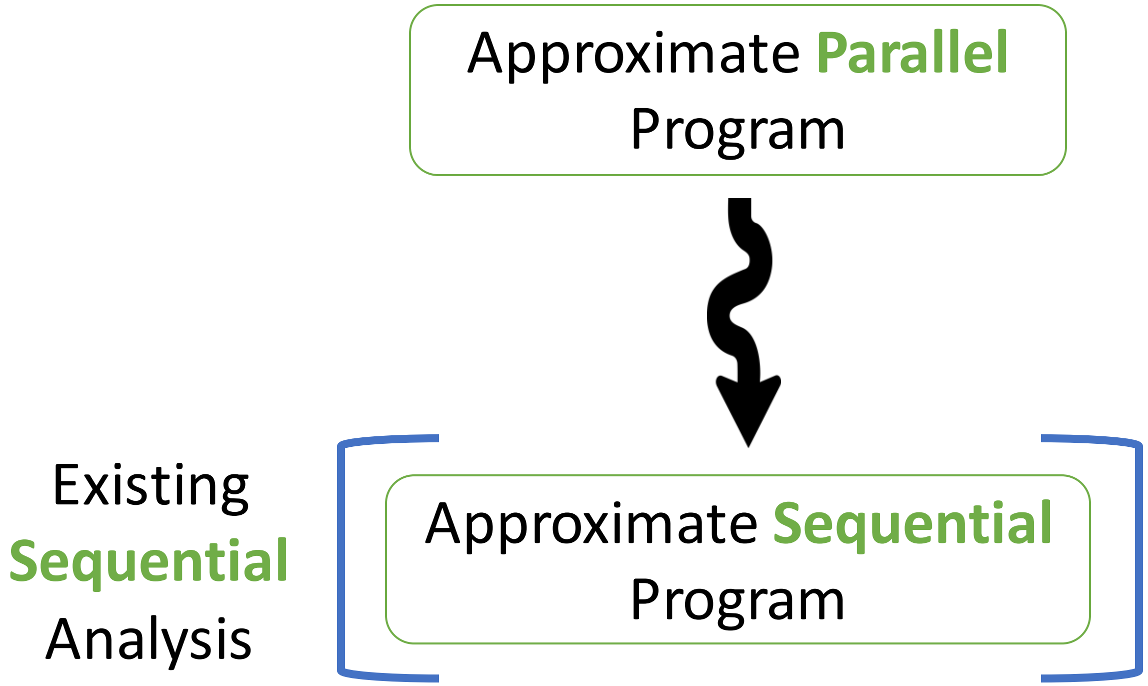
Existing **Sequential** Analysis

Approximate **Sequential** Program

How do we express parallel approximations?

How to enforce and verify safety/accuracy properties?

Under what conditions will the existing analyses apply?



Approximate **Parallel** Program

Existing **Sequential** Analysis

Approximate **Sequential** Program

Parallely!

Language with support for modeling parallel approximations

- Software-level approximation
- Environment-level noise

Verification of safety and accuracy using canonical sequentialization

- Type-safety (Non-Interference)
- Deadlock-freeness
- Relative safety
- Reliability
- Accuracy
- And more

Programs in Parallely

Asynchronous distributed message passing processes

Two types of data : **precise** and **approx.**

Communicates through **typed channels**

```
0:  send(1, precise int, input)
    out = receive(1, approx int)
```

||

```
1:  a = receive(0, precise int)
    result = computation(a)
    send(0, approx int, result)
```

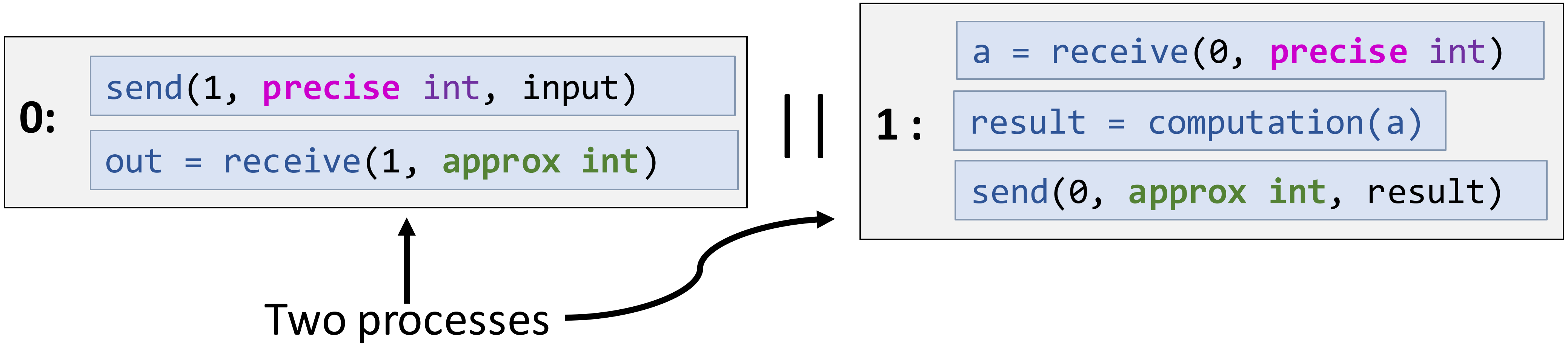
Programs in Parallely

```
0:  send(1, precise int, input)
    out = receive(1, approx int)
```

||

```
1:  a = receive(0, precise int)
    result = computation(a)
    send(0, approx int, result)
```

Programs in Parallely



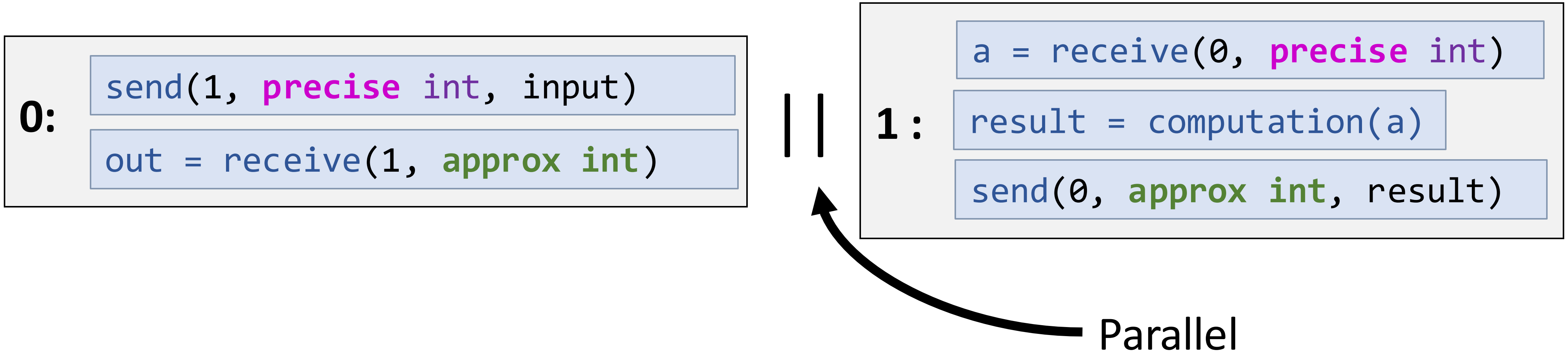
```
0:  send(1, precise int, input)
    out = receive(1, approx int)
```

||

```
1:  a = receive(0, precise int)
    result = computation(a)
    send(0, approx int, result)
```

Two processes

Programs in Parallely



```
0: send(1, precise int, input)
   out = receive(1, approx int)
```

||

```
1: a = receive(0, precise int)
   result = computation(a)
   send(0, approx int, result)
```

Parallel

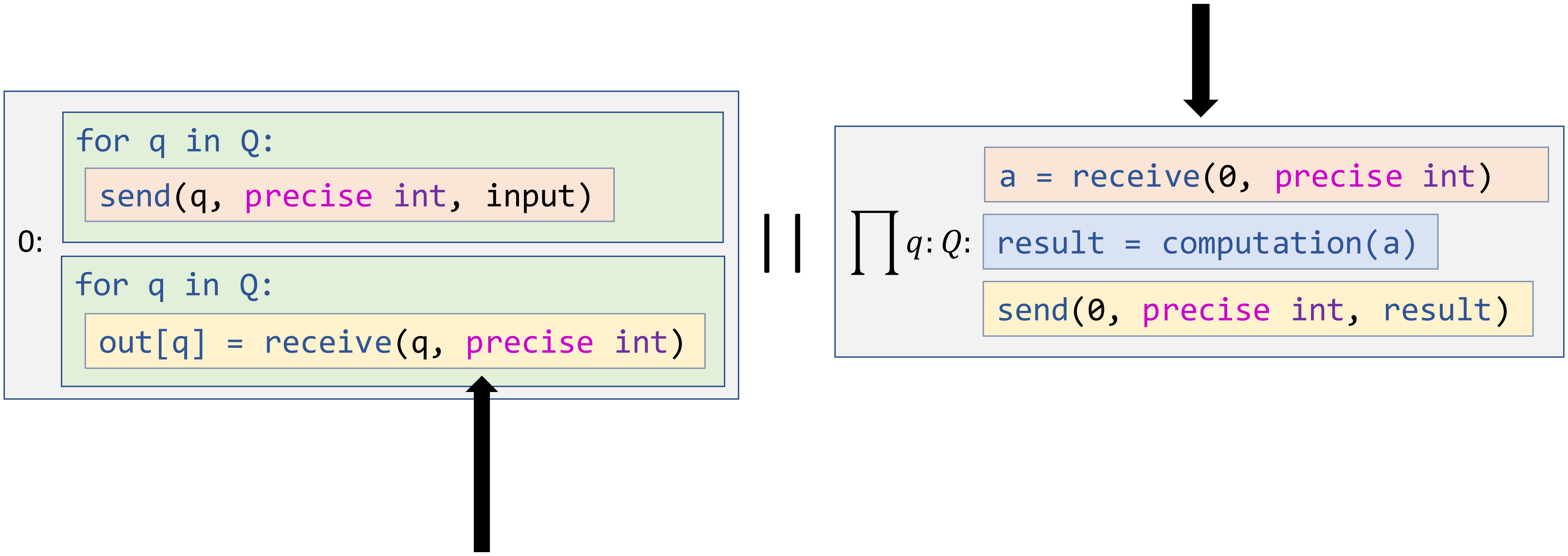
Programs in Parallely

```
0: send(1, precise int, input)
   out = receive(1, approx int)
```

||

```
1: a = receive(0, precise int)
   result = computation(a)
   send(0, approx int, result)
```

Symmetric Process Groups



Group of processes

```
0:
for q in Q:
    send(q, precise int, input)
for q in Q:
    out[q] = receive(q, precise int)
```

||

$\prod q : Q$:

```
a = receive(0, precise int)
result = computation(a)
send(0, precise int, result)
```

Iteration over a group of processes

Symmetric Non-determinism

All **receive** statements have a unique matching **send** statement

[Bakst et al. OOPSLA 2017]

Map-Reduce Pattern

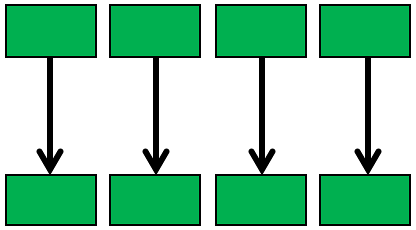
```
0:  for q in Q:
        send(q, precise int, input)
    for q in Q:
        out[q] = receive(q, precise int)
```

||

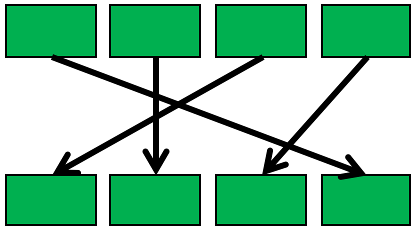
$\prod q : Q$:

```
a = receive(0, precise int)
result = computation(a)
send(0, precise int, result)
```

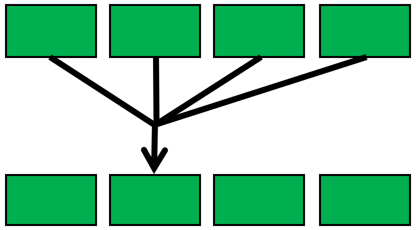
Communication Patterns easily expressible in Parallely



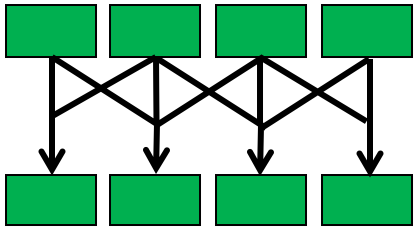
Map



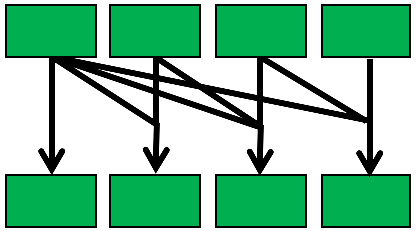
Scatter/Gather



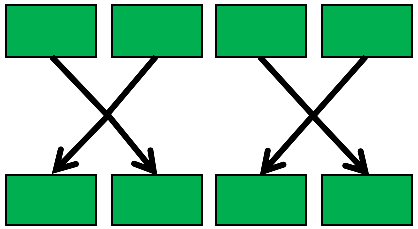
Reduce



Stencil



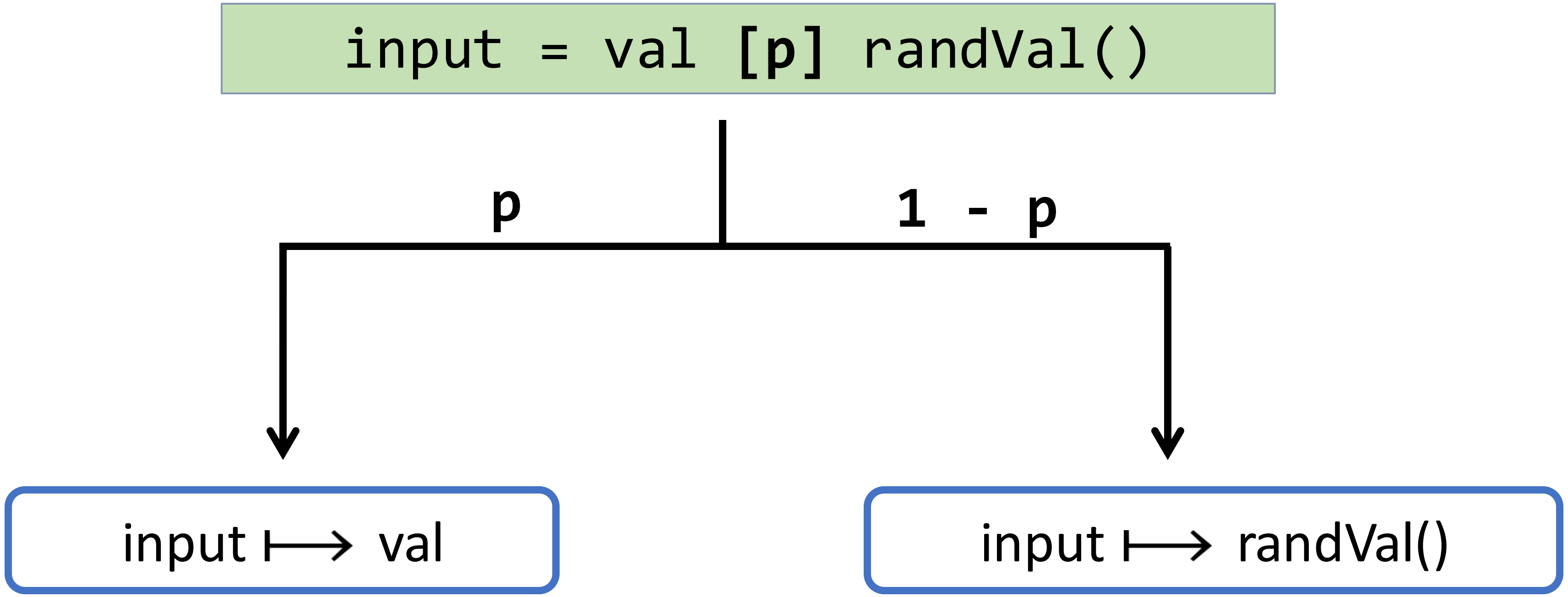
Scan



Partition

Covers all the patterns in [M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. 2014. Paraprox: Pattern-based Approximation for Data Parallel Applications. In ASPLOS.]

Approximation Primitives– Probabilistic Choice



```
input = val [p] randVal()
```

p

1 - p

input ⟼ val

input ⟼ randVal()

Approximation Primitives– Probabilistic Choice

```
input = val [p] randVal()
```

- Low energy channels that may corrupt the data being transmitted

```
0: input = val [p] randVal()
   send(1, approx int, input)    || 1: a = receive(0, approx int)
```

Approximation Primitives- Precision Conversion

- Casting to reducing the precision of data that has primitive numeric types

```
sVal = (approx float32) val
```

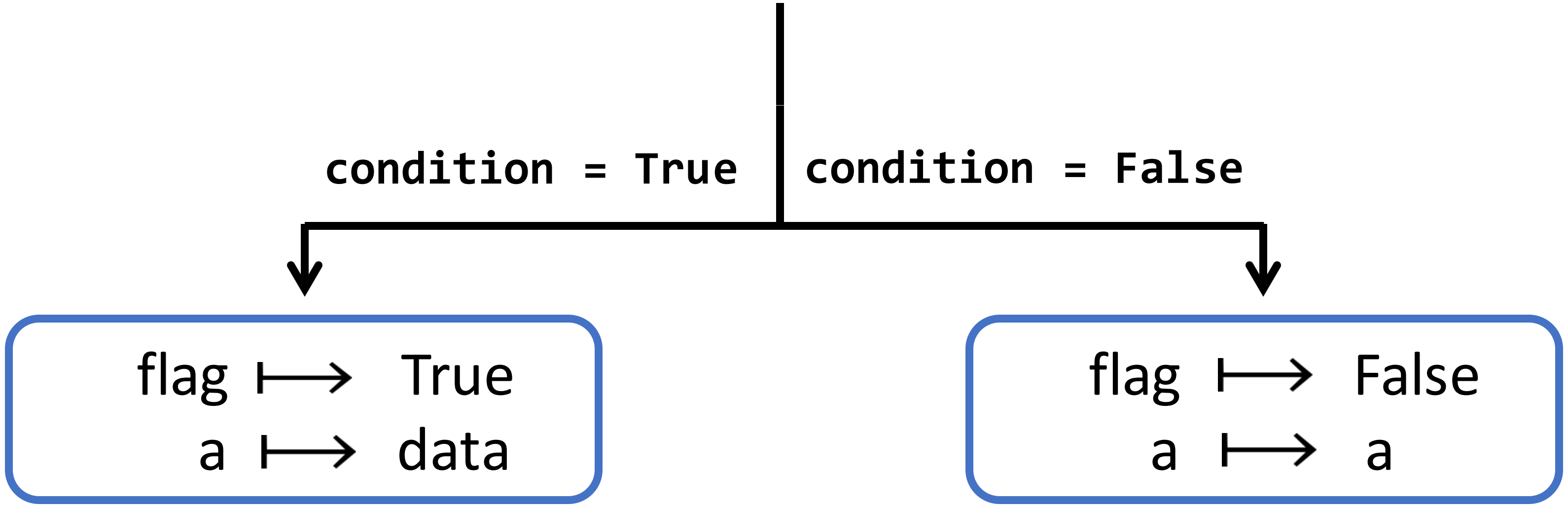
- Communicate in low precision

```
   sVal = (approx float32) val           tmp = receive(0, approx float32)
0:                                 || 1 :
   send(1, approx float32, sVal)         a = (approx float64) tmp
```

Approximation Primitives – Conditional Communication

```
0: cond-send(condition, 1, approx int, data)
```

```
1: flag, a = cond-receive(0, approx int)
```



Approximation Primitives – Conditional Communication

```
0: cond-send(condition, 1, approx int, data)
```

```
1: flag, a = cond-receive(0, approx int)
```

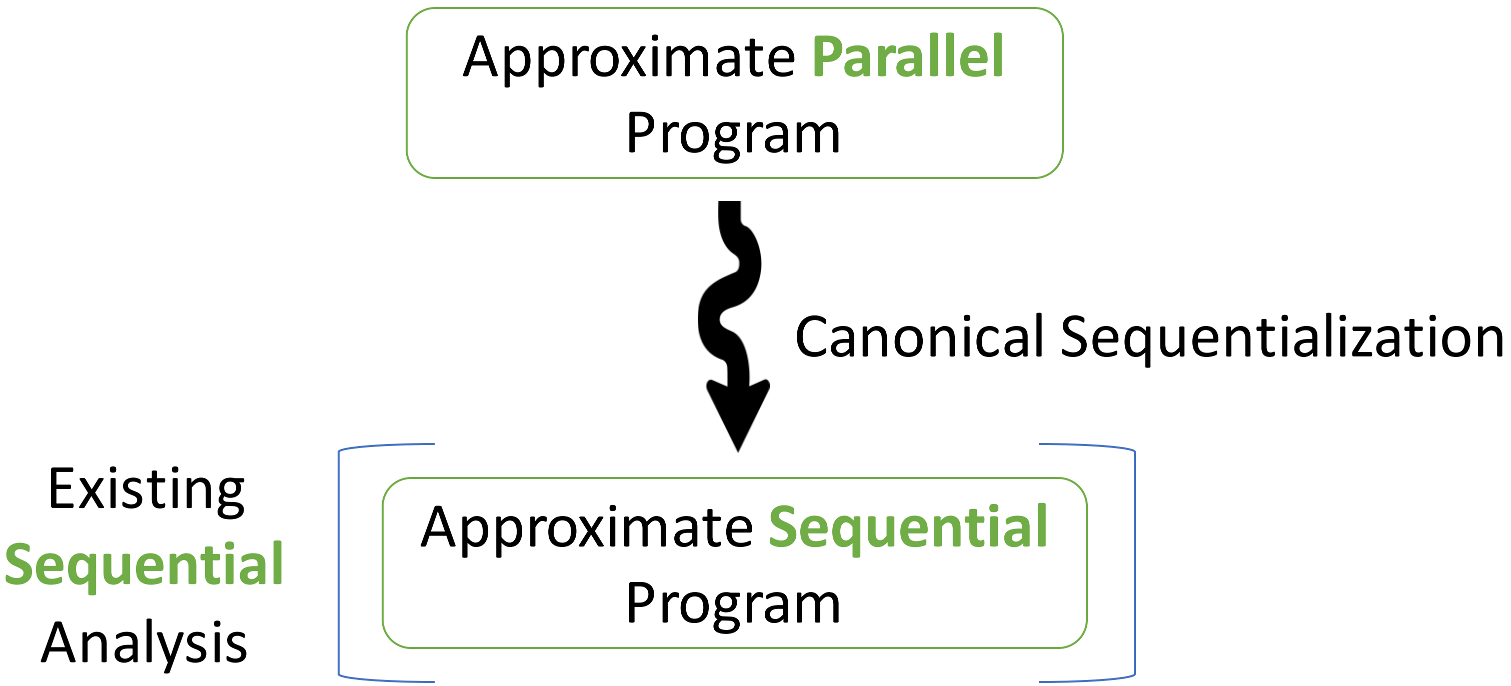
- Skip sending some data

```
    skip = 1 [0.99] 0
0:                                           || 1 : flag, a = cond-receive(0, approx int)
    cond-send(skip, 1, approx int, data)
```

What approximations can be modelled with Parallely

- Failing tasks – probabilistic-choice + conditional communication
- Noisy channel – probabilistic-choice
- Precision reduction – casting
- Memoization – probabilistic-choice + conditional communication
- Approximate reduce – probabilistic-choice + conditional communication
- Loop perforation – probabilistic-choice

How do we analyze Parallely programs?



Approximate **Parallel** Program

Canonical Sequentialization

Approximate **Sequential** Program

Existing **Sequential** Analysis

Canonical Sequentialization (Bakst et al. OOPSLA 2017)

Generate an equivalent sequential program using rewriting

Works for programs with **symmetric nondeterminism**

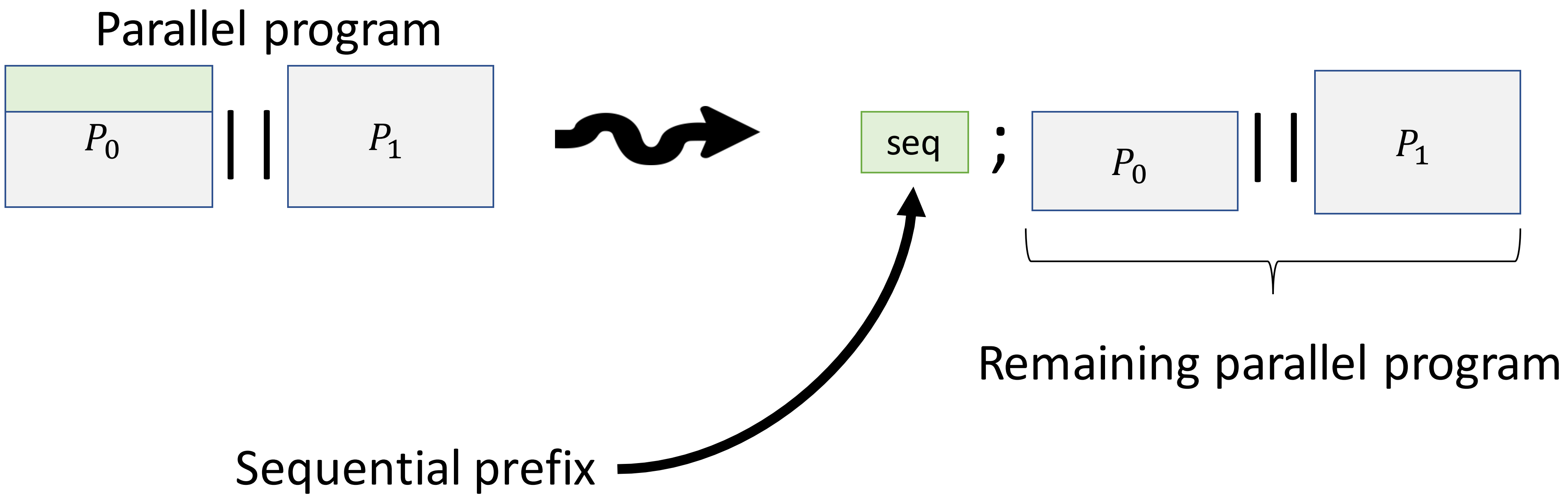
We show how *sequentialization* works for

Probabilistic choice
x = y **[p]** z

Casting
x = **(float32)** y

Conditional Communication
cond-send(b, tid, type, val)

Sequentialization through rewrites



Parallel program

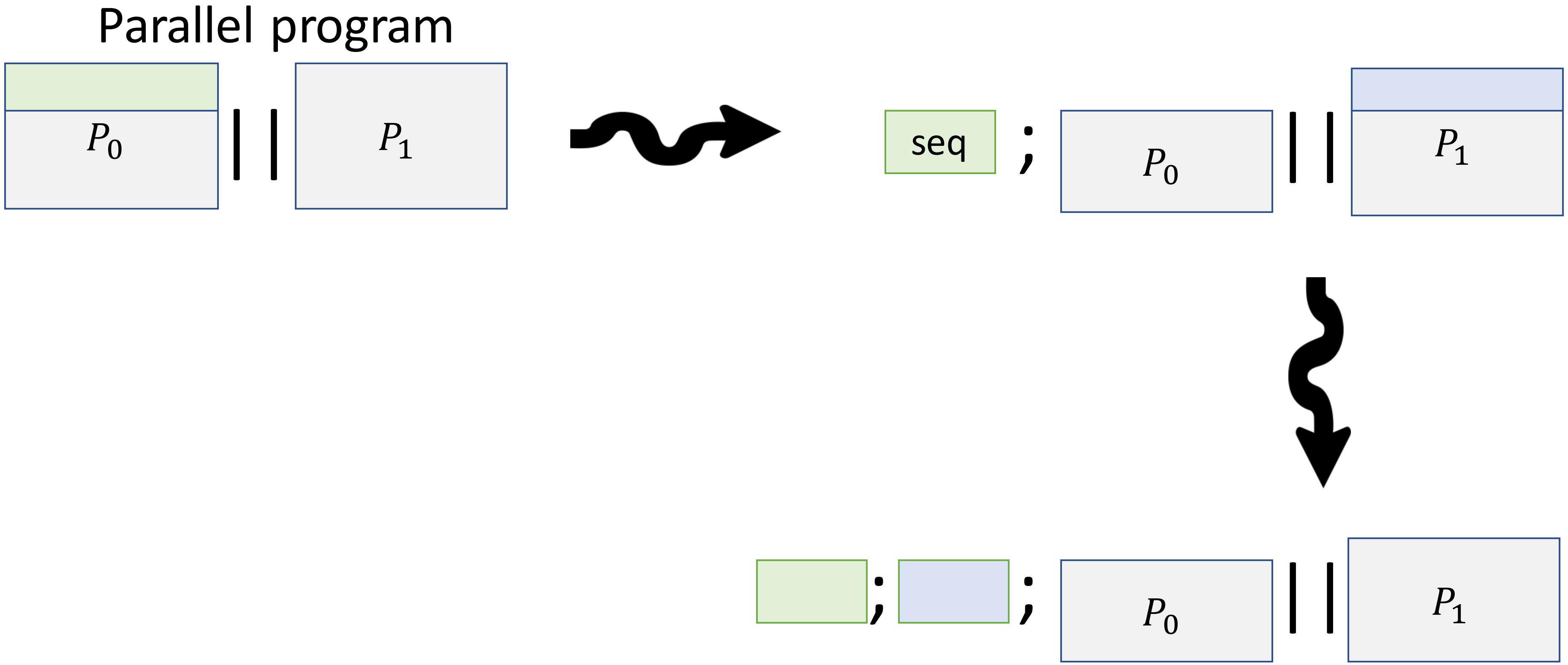
P_0 || P_1

seq ; P_0 || P_1

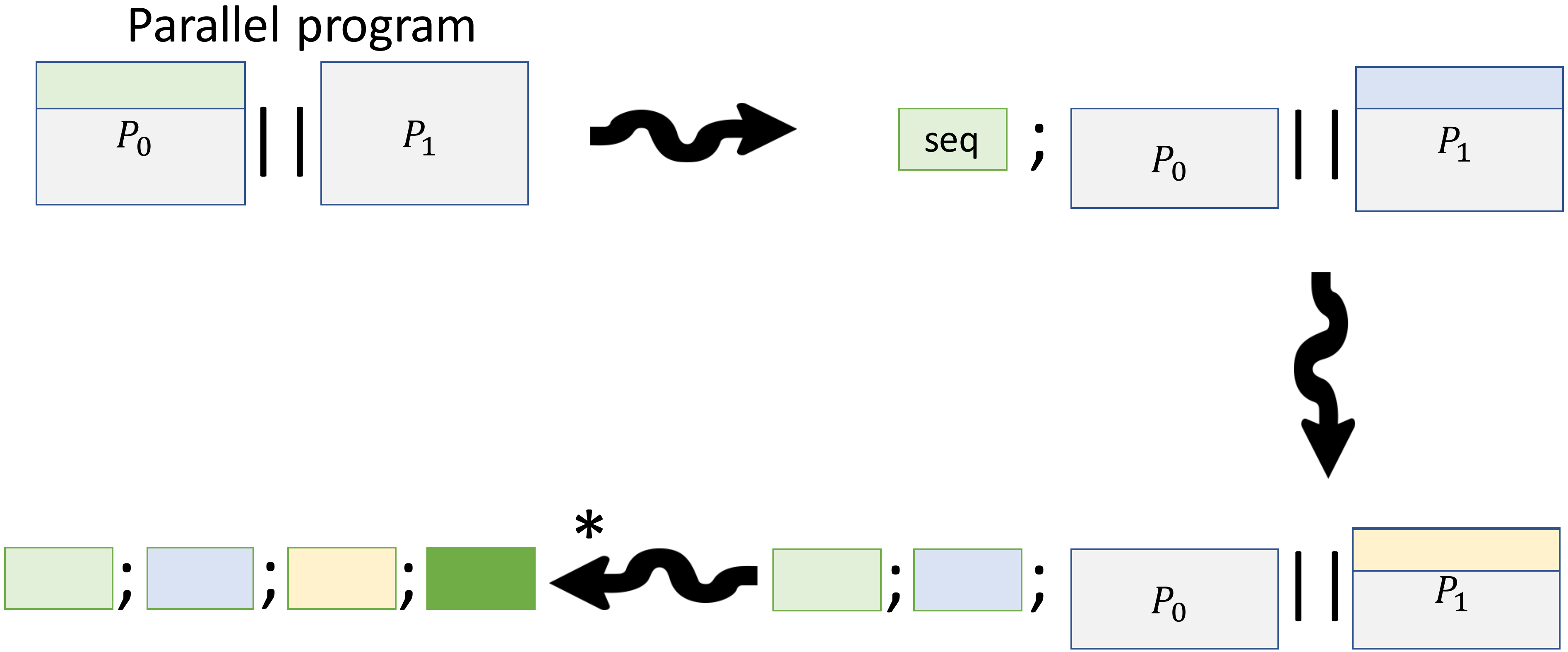
Sequential prefix

Remaining parallel program

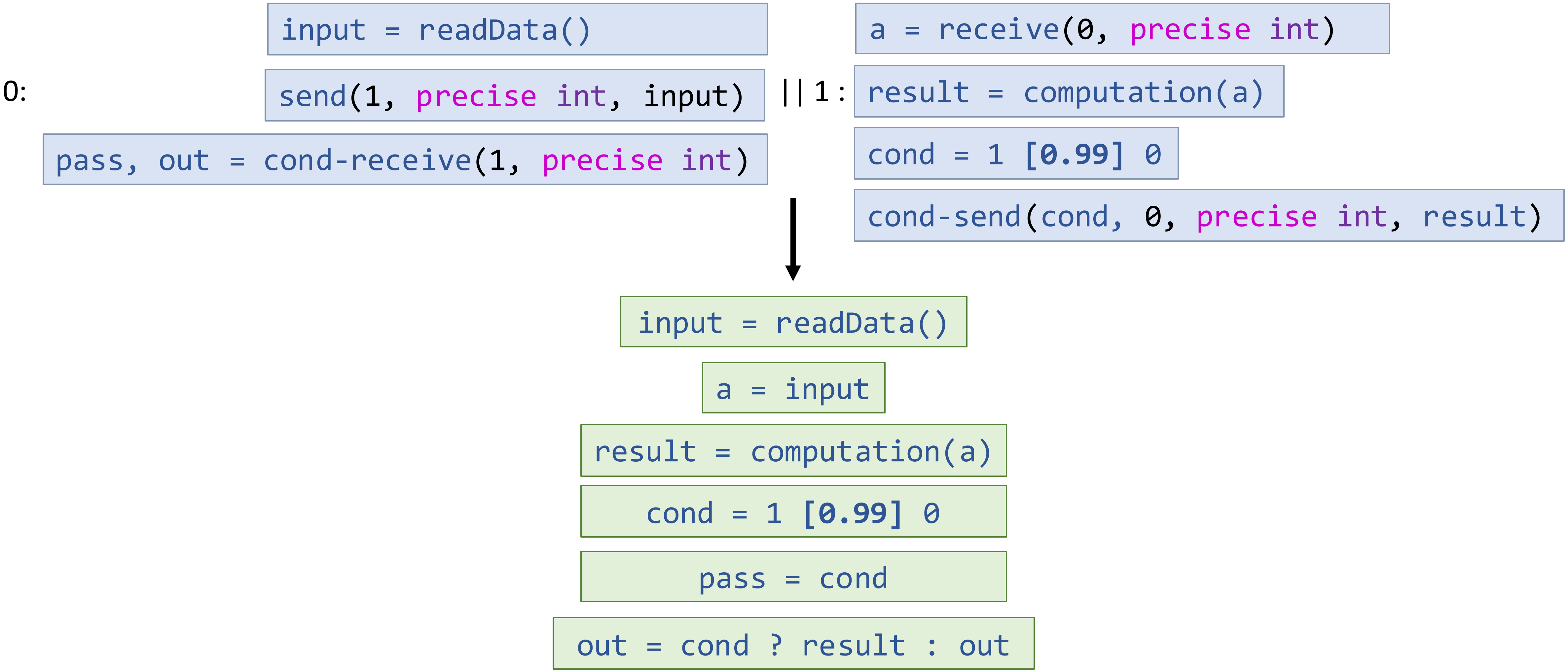
Sequentialization through rewrites



Sequentialization through rewrites



Generating a Canonical Sequentialization



0:

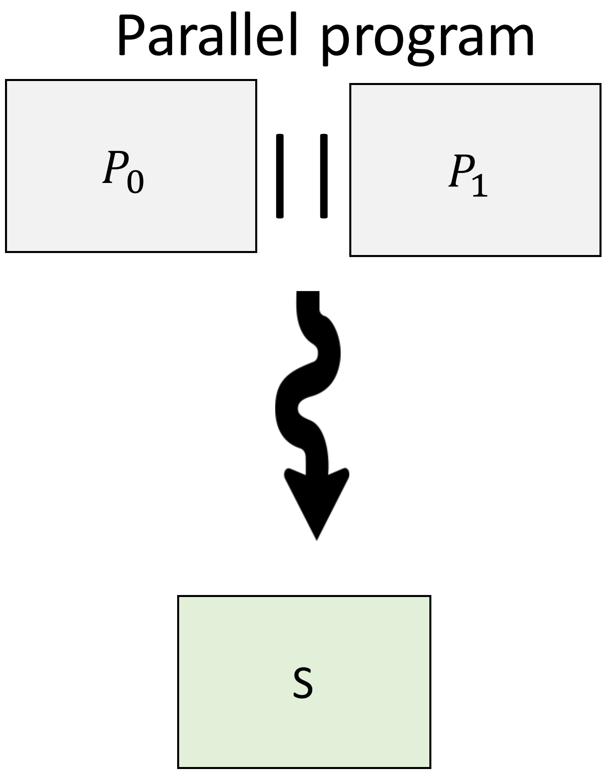
```
input = readData()
send(1, precise int, input)
pass, out = cond-receive(1, precise int)
```

|| 1 :

```
a = receive(0, precise int)
result = computation(a)
cond = 1 [0.99] 0
cond-send(cond, 0, precise int, result)
```

```
input = readData()
a = input
result = computation(a)
cond = 1 [0.99] 0
pass = cond
out = cond ? result : out
```

Rewrite Soundness – Intuition

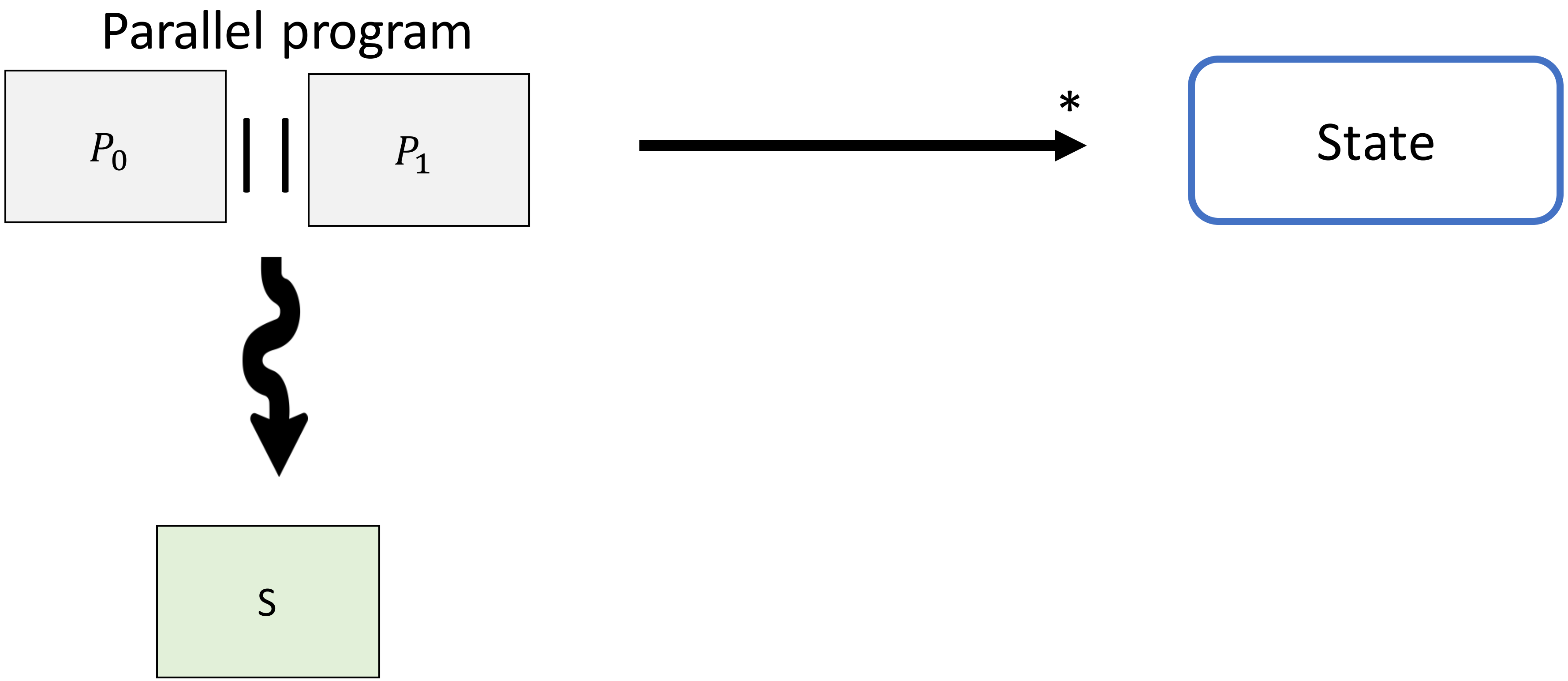


Parallel program

P_0 || P_1

S

Rewrite Soundness – Intuition



Parallel program

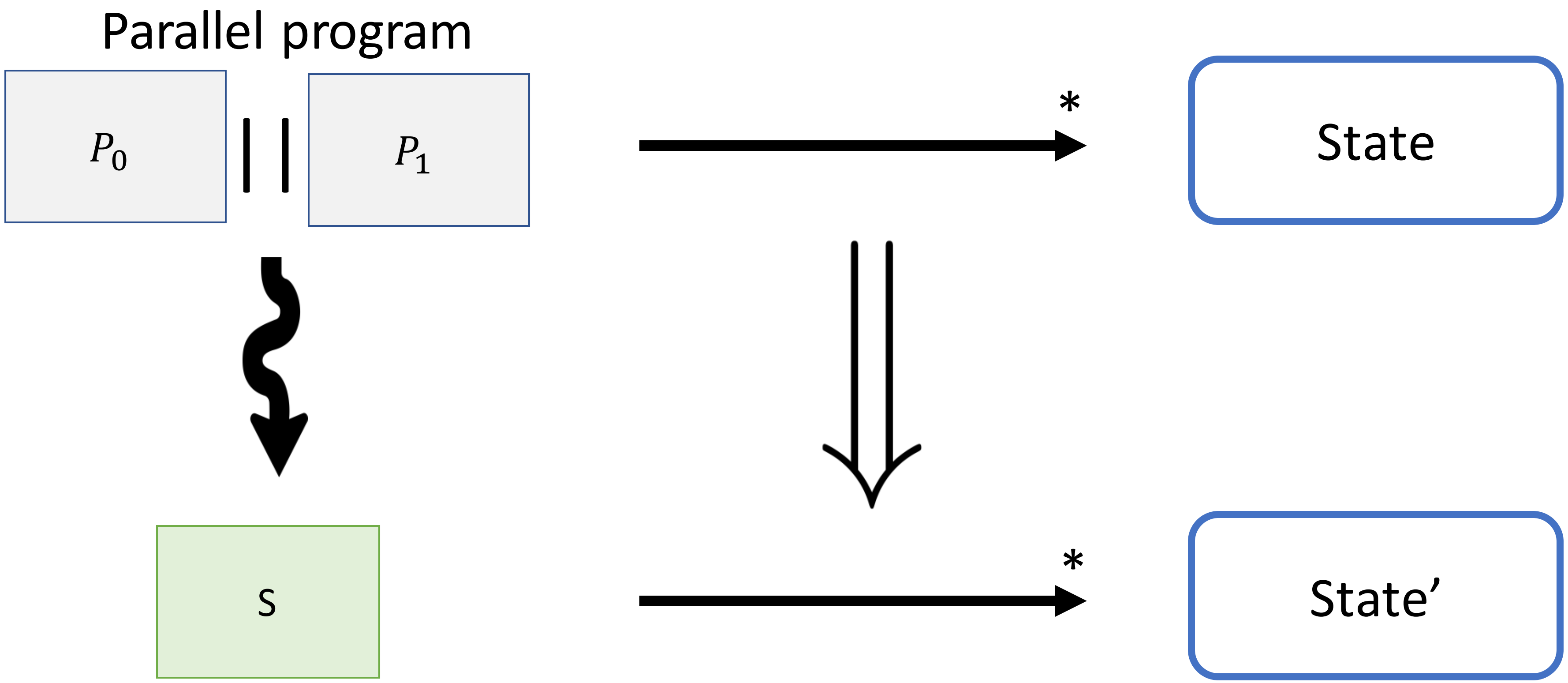
$P_0 \,||\, P_1$

$*$

State

S

Rewrite Soundness – Intuition



Parallel program

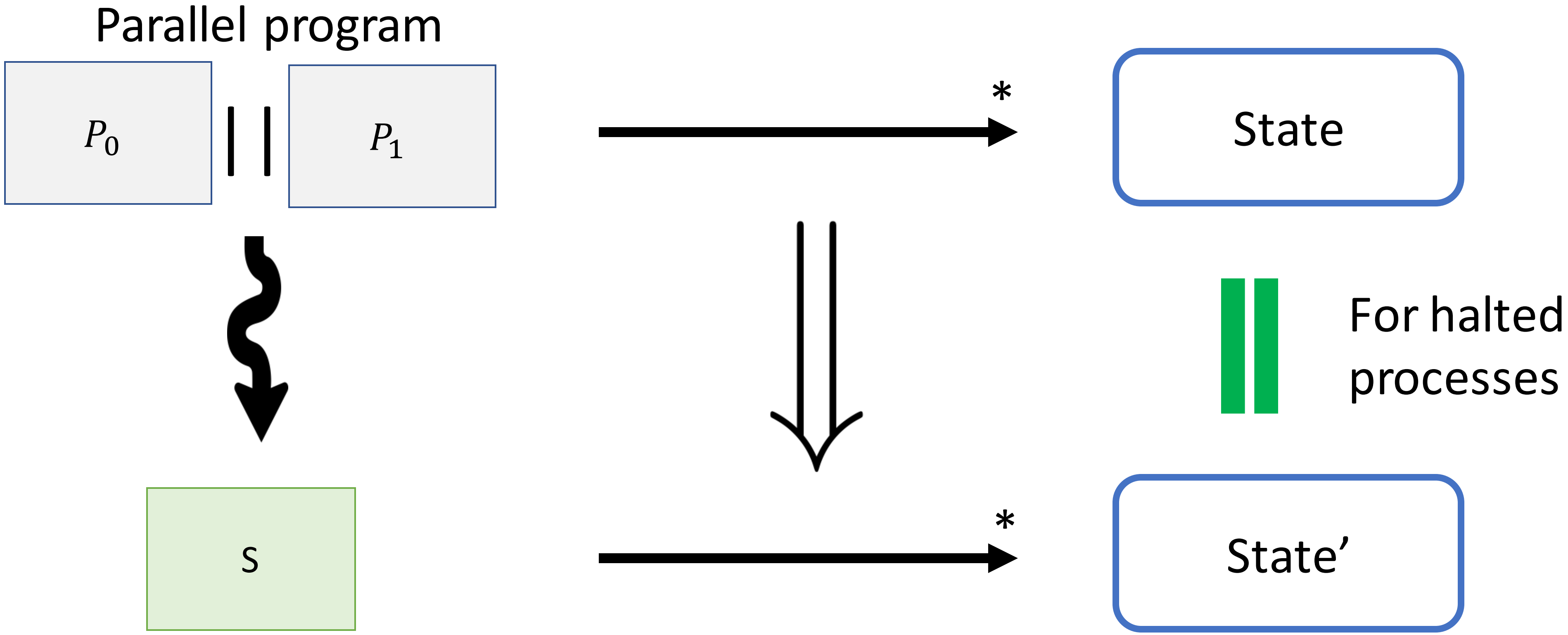
P_0 || P_1

S

State

State’

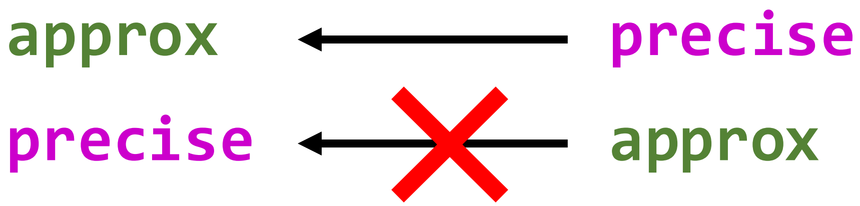
Rewrite Soundness – Intuition



Parallel program

Non-Interference

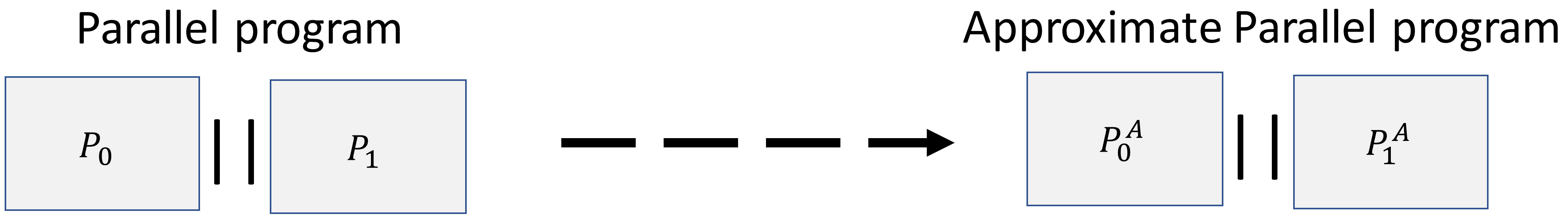
- Set of type rules that block explicit and implicit flows in each individual process



- Typed channels and sequentialization detects illegal flows across process boundaries

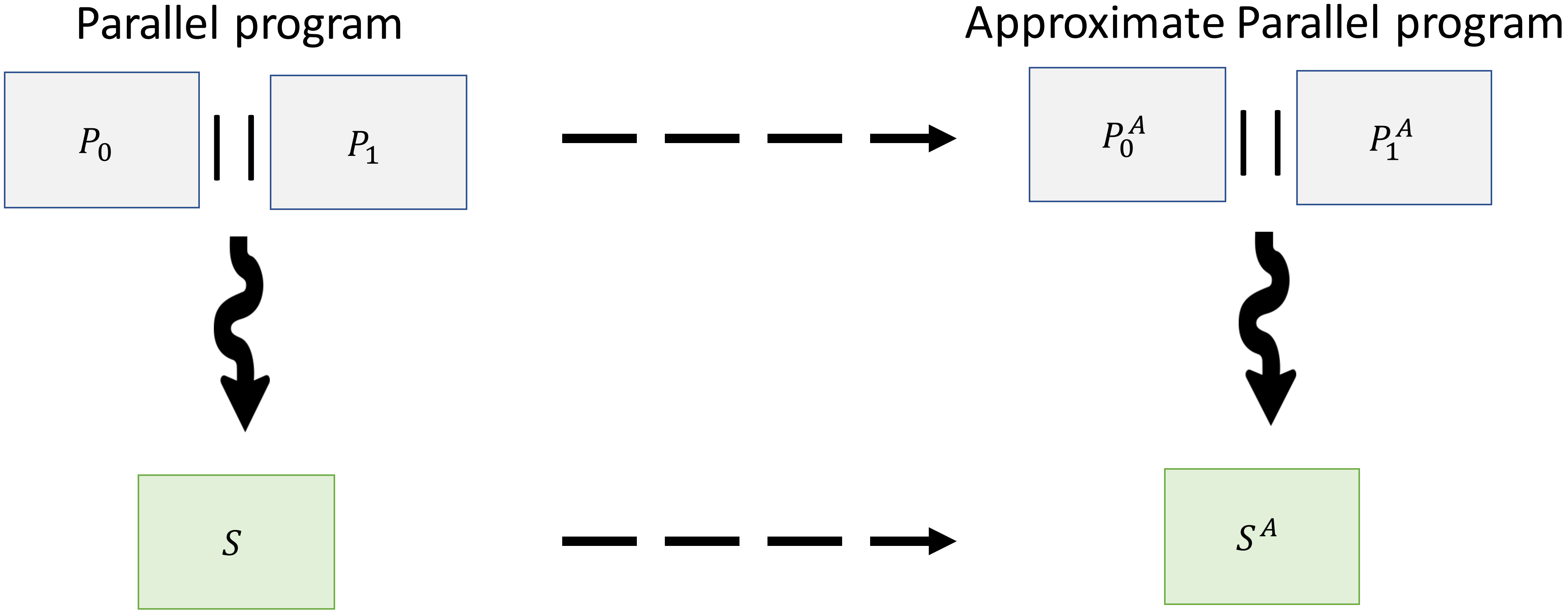
```
0: send(1, approx int, result)   ||   1: out = receive(0, precise int)
```

Relative Safety



If the **original program** satisfies a **property**, then the **transformed program** also satisfies that property

Relative Safety



We can use the sequentialized programs to prove relative safety for process local safety property

Program type checks

+

There is a canonical sequentialization

→

No Deadlocks
(Bakst et al. OOPSLA 2017)

Non-interference

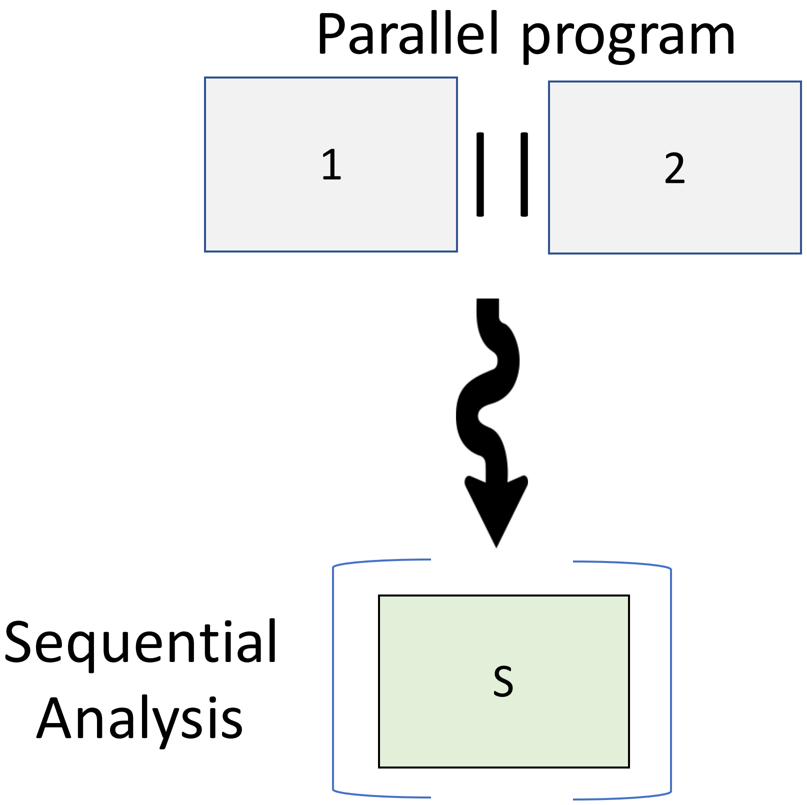
Relative safety

Reliability/Accuracy analysis

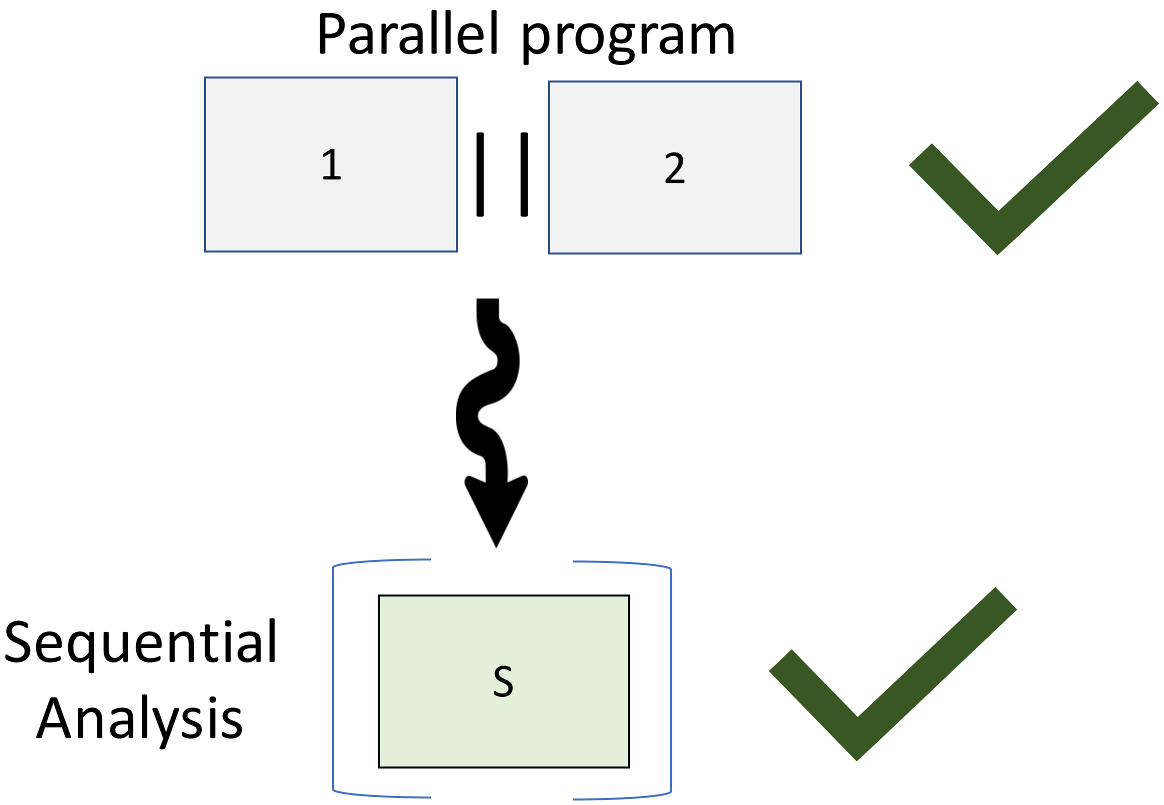
Reliability – Probability that an approximate execution produces the same result as an exact one

Accuracy – Probability that an approximate execution produces a result close to an exact one

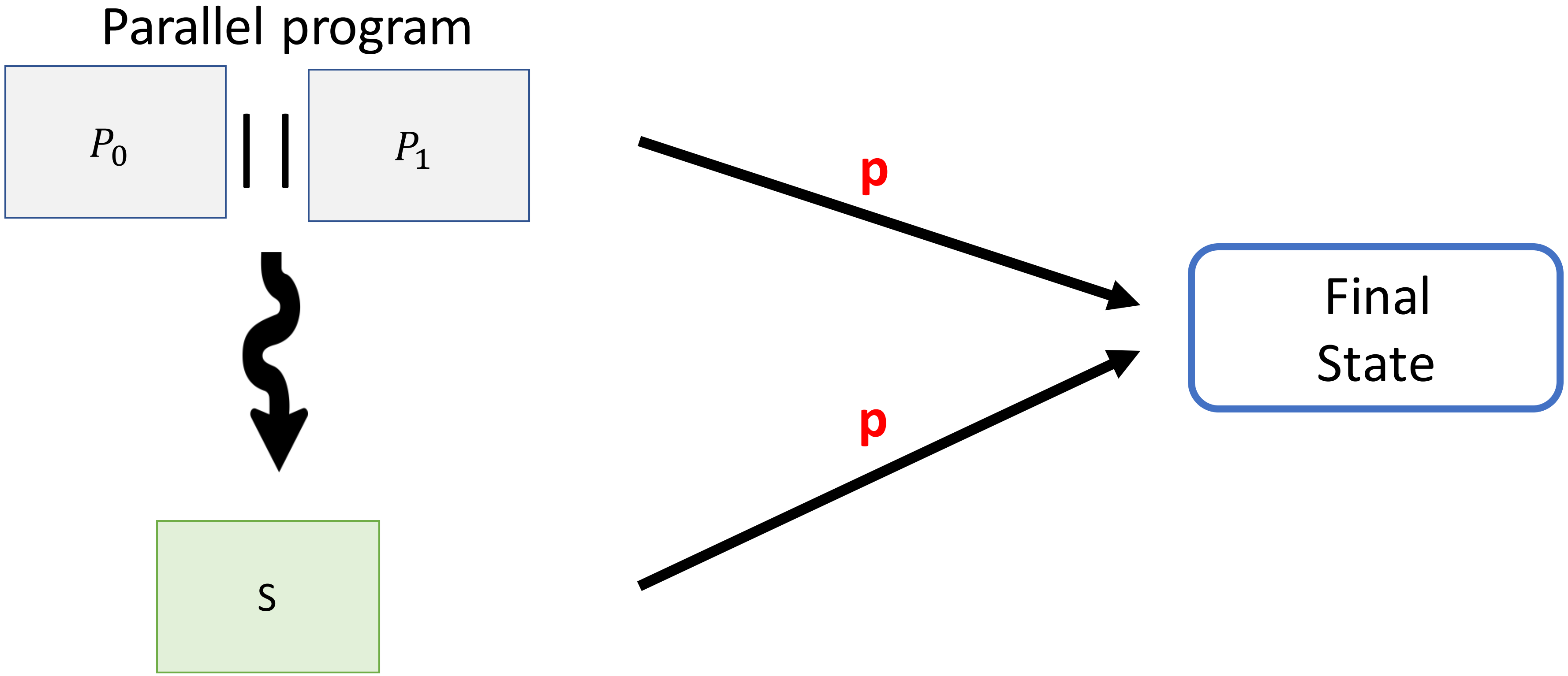
Reliability/Accuracy analysis



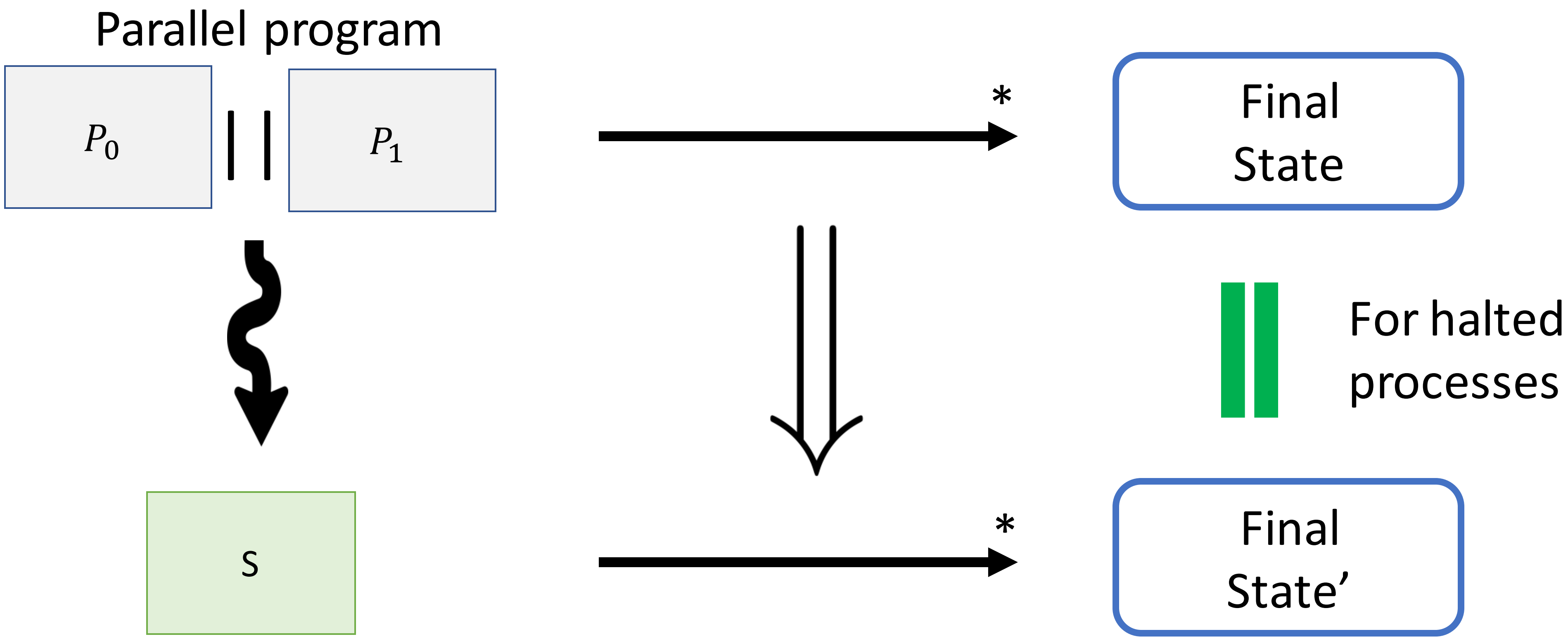
Reliability/Accuracy analysis



Rewrite Equivalence



Rewrite Soundness



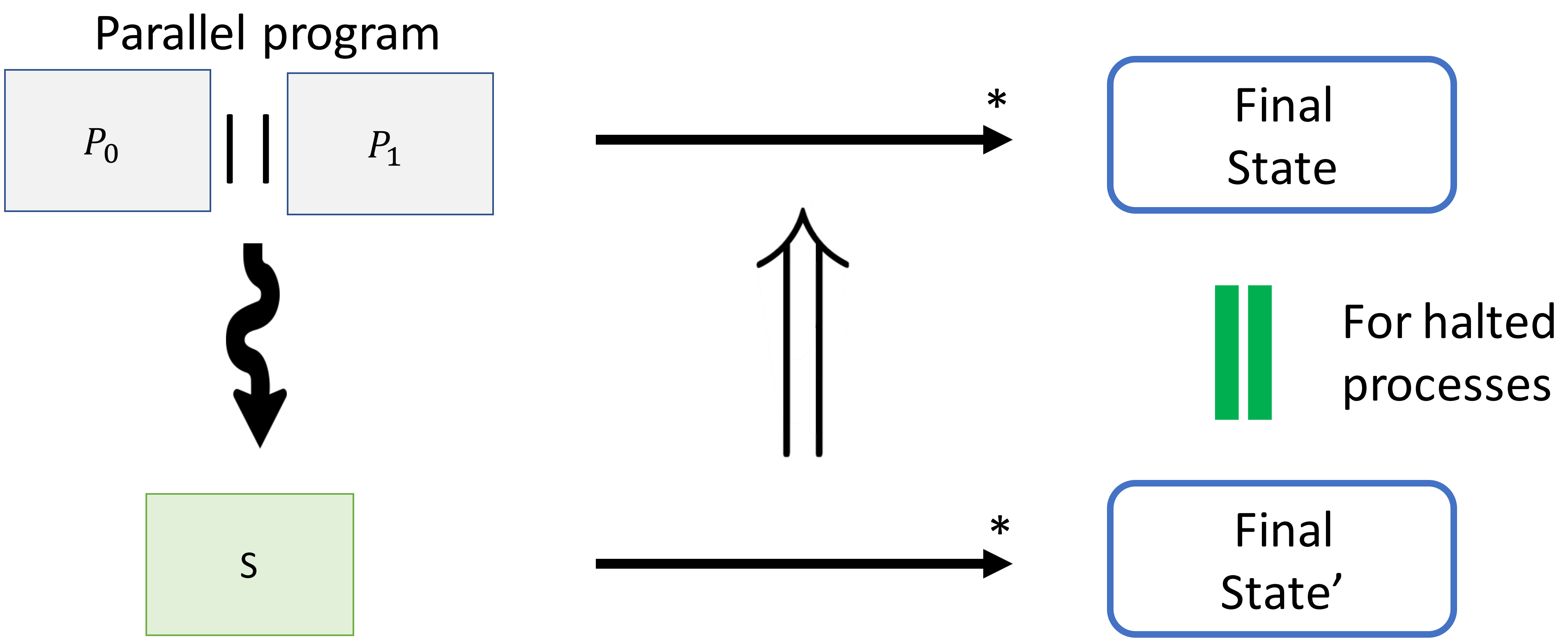
Parallel program

P_0 || P_1 $\xrightarrow{}^{*}$ Final State

S $\xrightarrow{}^{*}$ Final State’

For halted processes

Rewrite Equivalence



Parallel program

P_0 || P_1

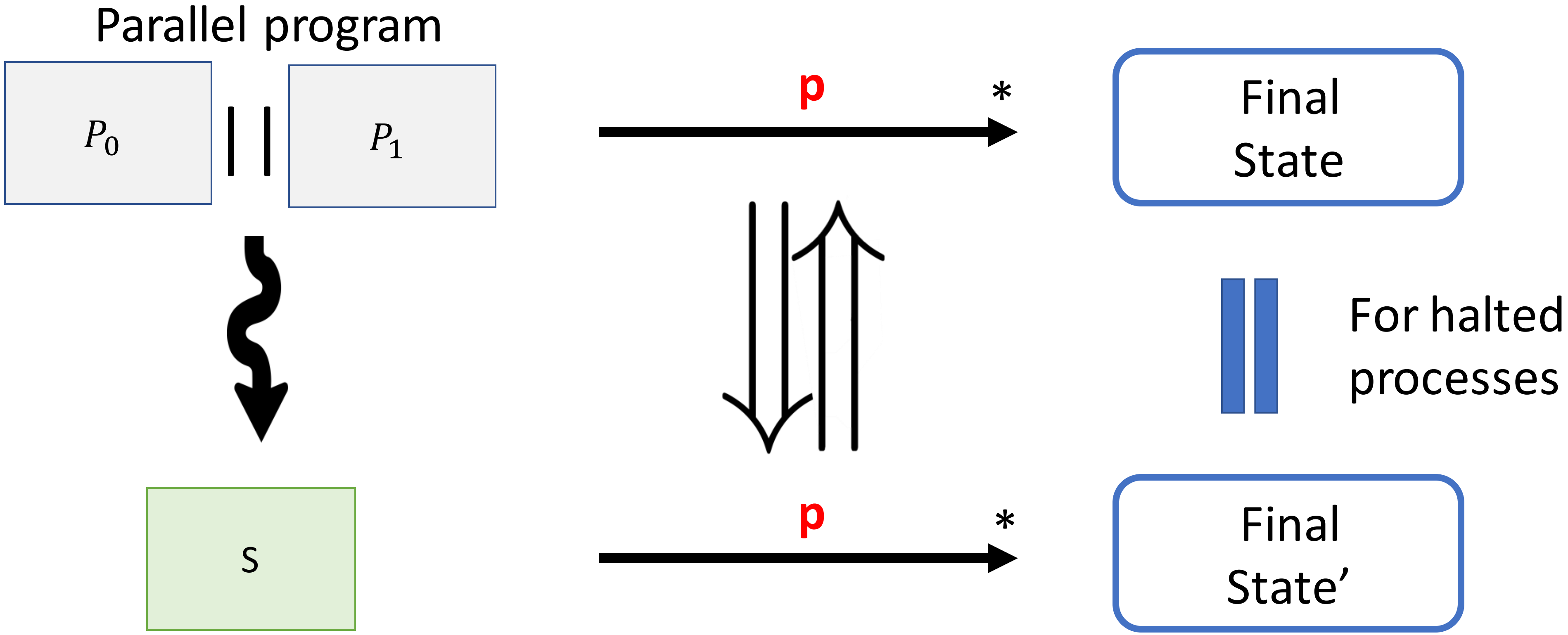
S

Final State

Final State’

For halted processes

Rewrite Equivalence



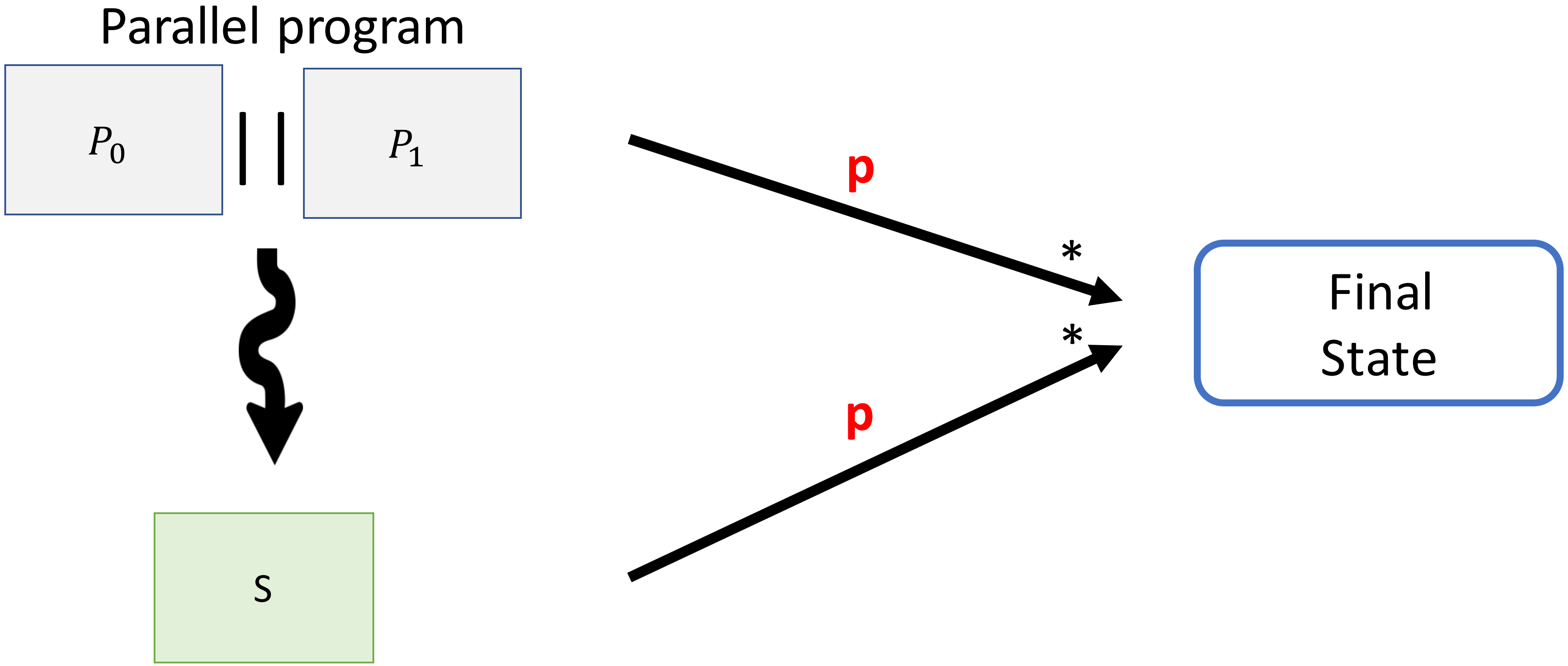
Parallel program

P_0 || P_1 $\xrightarrow{\mathbf{p}}{}^{*}$ Final State

S $\xrightarrow{\mathbf{p}}{}^{*}$ Final State’

For halted processes

Rewrite Equivalence

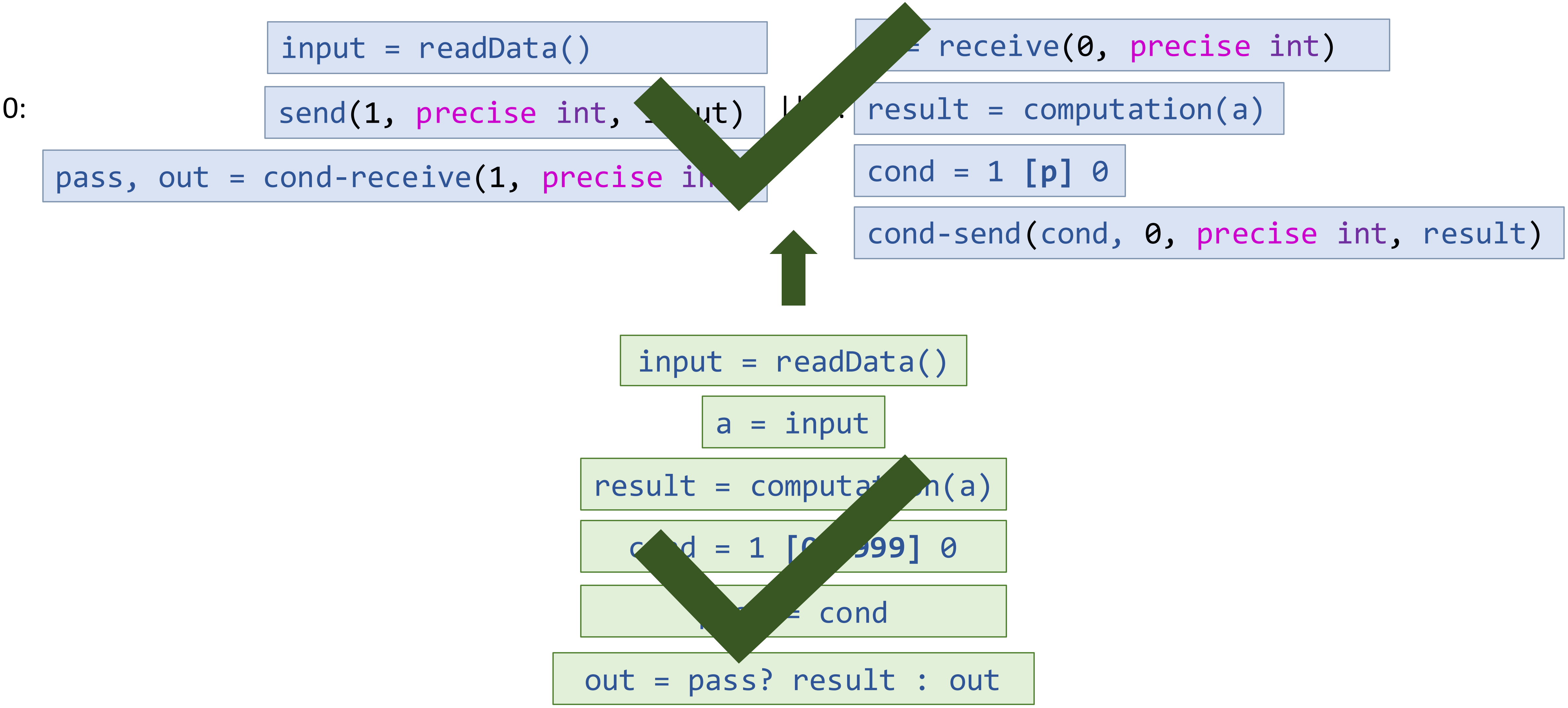


Reliability Analysis (Rely – Carbin et al. 2013)

$$0.99 \leq \ R(out)$$

```
input = readData()
a = input
result = computation(a)
cond = 1 [0.999] 0
pass = cond
out = pass? result : out
```

Reliability Analysis



```
0:    input = readData()
      send(1, precise int, i   ut)
      pass, out = cond-receive(1, precise in
```

```
||
```

```
   = receive(0, precise int)
result = computation(a)
cond = 1 [p] 0
cond-send(cond, 0, precise int, result)
```

```
input = readData()
a = input
result = computa   n(a)
c   d = 1 [0.  999] 0
p    = cond
out = pass? result : out
```

Program type checks

+

There is a canonical sequentialization

→

No Deadlocks

Non-interference

Relative safety

Reliability and accuracy analysis on the sequential program valid on the parallel

Evaluation - Benchmarks

Benchmark	Parallel Pattern	Approximation
PageRank	Map	Failing Tasks
Scale	Map	Failing Tasks
Blackscholes	Map	Noisy Channel
SSSP	Scatter-Gather	Noisy Channel
BFS	Scatter-Gather	Noisy Channel
SOR	Stencil	Precision Reduction
Motion	Map/Reduce	Approximate Reduce
Sobel	Stencil	Precision Reduction

Benchmarks – Verification Time

Benchmark	Approximation	Property	Time	
			Type + Seq	Rel / Acc
PageRank	Failing Tasks	Safety + Reliability (0.99)	1.8s	168s
Scale	Failing Tasks	Safety + Reliability (0.99)	6.5s	7.4s
Blackscholes	Noisy Channel	Safety + Reliability (0.99)	0.2s	12s
SSSP	Noisy Channel	Safety + Reliability (0.99)	9.6s	9.6s
BFS	Noisy Channel	Safety + Reliability (0.99)	8.9s	9.2s
SOR	Precision Reduction	Safety + Accuracy bound (10^{-6})	8.3s	53s
Motion	Approx Reduce	Safety	3.9s	-
Sobel	Precision Reduction	Safety + Accuracy bound (10^{-6})	0.2s	72s

Also in the paper

- Evaluation of the benefits of approximations

- Type System and Proof for non-interference

- Soundness Proofs for reliability and accuracy analysis

New Directions

- Generalizing to verification of other properties – fairness
- Dynamic analysis – proving correctness of runtime systems
- Other parallel models – shared memory, etc

Takeaways

- Parallely is a language that can express many common approximation patterns through three simple approximation primitives

- Parallely leverages canonical sequentialization to extend many existing and future analyses from sequential to parallel programs

- Efficiently verifies safety and accuracy of 8 kernels and 8 popular approximate computing benchmarks