Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization

Vimuth Fernando, Keyur Joshi, Sasa Misailovic
University of Illinois at Urbana-Champaign
Compression

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
• **Types** – Non-interference of approximate and precise data [Sampson et al. 2011]

• **Relative safety** – Transfer reasoning about original programs 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]
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

Approximate Sequential Program

Existing Sequential Analysis
How do we express parallel approximations?

How to enforce and verify safety/accuracy properties?

Under what conditions will the existing analyses apply?
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)`
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

for q in Q:
    send(q, precise int, input)

for q in Q:
    out[q] = receive(q, precise int)

\[ a = \text{receive}(\theta, \text{precise int}) \]
\[ \prod_{q \in Q} \text{result} = \text{computation}(a) \]
\[ \text{send}(\theta, \text{precise int}, \text{result}) \]

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)

\(\prod_{q \in Q} \) for q in Q:

\(a = receive(\emptyset, \text{precise int})\)

\(result = \text{computation}(a)\)

\(send(\emptyset, \text{precise int, result})\)
Communication Patterns easily expressible in Parallely

- Map
- Scatter/Gather
- Reduce
- Stencil
- Scan
- Partition

Approximation Primitives—Probabilistic Choice

\[
\text{input} = \text{val}[p] \text{ randVal()}
\]
Approximation Primitives– Probabilistic Choice

- Low energy channels that may corrupt the data being transmitted

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

0: 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

\[
\text{sVal} = (\text{approx float32}) \text{ val}
\]

• Communicate in low precision

0: \[
\begin{align*}
\text{sVal} &= (\text{approx float32}) \text{ val} \\
\text{send}(1, \text{approx float32}, \text{sVal})
\end{align*}
\]

1: \[
\begin{align*}
\text{tmp} &= \text{receive}(0, \text{approx float32}) \\
\text{a} &= (\text{approx float64}) \text{ tmp}
\end{align*}
\]
Approximation Primitives – Conditional Communication

0: \texttt{cond-send(\textit{condition}, 1, approx int, data)}

1: \texttt{flag, a = cond-receive(0, approx int)}

- condition = True
  - flag \rightarrow True
  - a \rightarrow data

- condition = False
  - flag \rightarrow False
  - a \rightarrow a
Approximation Primitives – Conditional Communication

0: \textbf{cond-send}(condition, 1, \textbf{approx int}, data)

1: flag, a = \textbf{cond-receive}(\emptyset, \textbf{approx int})

• Skip sending some data

\begin{align*}
\text{skip} & = 1 \ [0.99] \ 0 \\
\text{cond-send}(\text{skip}, 1, \textbf{approx int}, \text{data}) & \quad || \quad 1: \text{flag, a} = \textbf{cond-receive}(\emptyset, \textbf{approx int})
\end{align*}
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 Parallelly 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 \textit{symmetric nondeterminism}

We show how \textit{sequentialization} works for

- Probabilistic choice: $x = y \ [p] \ z$
- Casting: $x = \text{(float32)} \ y$
- Conditional Communication: \text{cond-send}(b, \text{tid, type, val})
Sequentialization through rewrites

Parallel program

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

\[ P_0 \parallel P_1 \]

S
Rewrite Soundness – Intuition

Parallel program

\[ P_0 \parallel P_1 \]

\[ S \]

\[ * \]

State
Rewrite Soundness – Intuition

Parallel program

\[ P_0 \parallel P_1 \]

\[ S \]

\[ * \]

\[ \text{State} \]

\[ \text{State'} \]
Parallel program

$P_0$ || $P_1$

$s$

State

For halted processes

State'
Non-Interference

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

  \[
  \text{approx} \quad \leftrightarrow \quad \text{precise} \\
  \text{precise} \quad \leftrightarrow \quad \text{approx}
  \]

- Typed channels and sequentialization detects illegal flows across process boundaries

  \[
  0: \text{send}(1, \text{approx int, result}) \quad \| \quad 1: \text{out} = \text{receive}(0, \text{precise int})
  \]
Relative Safety

If the original program satisfies a property, then the transformed program also satisfies that property.
Relative Safety

Parallel program

\begin{align*}
P_0 & \quad \| \quad P_1
\end{align*}

Approximate Parallel program

\begin{align*}
P_0^A & \quad \| \quad P_1^A
\end{align*}

\begin{align*}
S & \quad \rightarrow \quad S^A
\end{align*}

We can use the sequentialized programs to prove relative safety for process local safety property
There is a canonical sequentialization

Program type checks +

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

Parallel program

1

2

Sequential Analysis

S
Reliability/Accuracy analysis
Rewrite Equivalence

Parallel program

\[ P_0 \parallel \underline{\underline{\parallel}} \underline{\underline{\parallel}} \underline{\underline{\parallel}} \underline{\underline{\parallel}} \underline{\underline{\parallel}} P_1 \]

\[ S \]

Final State
Rewrite Soundness

Parallel program

\[ P_0 \parallel P_1 \]

\[ S \]

\[ \star \]

For halted processes

Final State

Final State'
Rewrite Equivalence

Parallel program

$P_0 \parallel P_1$

For halted processes

$\text{Final State'}$

Final State
Rewrite Equivalence

Parallel program

$P_0 \parallel P_1$

For halted processes

Final State

Final State’
Rewrite Equivalence

Parallel program

$P_0 \parallel P_1$

$s$

Final State
Reliability Analysis (Rely – Carbin et al. 2013)

\[ 0.99 \leq R(\text{out}) \]
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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parallel Pattern</th>
<th>Approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td>Map</td>
<td>Failing Tasks</td>
</tr>
<tr>
<td>Scale</td>
<td>Map</td>
<td>Failing Tasks</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>Map</td>
<td>Noisy Channel</td>
</tr>
<tr>
<td>SSSP</td>
<td>Scatter-Gather</td>
<td>Noisy Channel</td>
</tr>
<tr>
<td>BFS</td>
<td>Scatter-Gather</td>
<td>Noisy Channel</td>
</tr>
<tr>
<td>SOR</td>
<td>Stencil</td>
<td>Precision Reduction</td>
</tr>
<tr>
<td>Motion</td>
<td>Map/Reduce</td>
<td>Approximate Reduce</td>
</tr>
<tr>
<td>Sobel</td>
<td>Stencil</td>
<td>Precision Reduction</td>
</tr>
<tr>
<td>Benchmark</td>
<td>Approximation</td>
<td>Property</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Type + Seq</td>
</tr>
<tr>
<td>PageRank</td>
<td>Failing Tasks</td>
<td>Safety + Reliability (0.99)</td>
</tr>
<tr>
<td>Scale</td>
<td>Failing Tasks</td>
<td>Safety + Reliability (0.99)</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>Noisy Channel</td>
<td>Safety + Reliability (0.99)</td>
</tr>
<tr>
<td>SSSP</td>
<td>Noisy Channel</td>
<td>Safety + Reliability (0.99)</td>
</tr>
<tr>
<td>BFS</td>
<td>Noisy Channel</td>
<td>Safety + Reliability (0.99)</td>
</tr>
<tr>
<td>SOR</td>
<td>Precision Reduction</td>
<td>Safety + Accuracy bound (10^{-6})</td>
</tr>
<tr>
<td>Motion</td>
<td>Approx Reduce</td>
<td>Safety</td>
</tr>
<tr>
<td>Sobel</td>
<td>Precision Reduction</td>
<td>Safety + Accuracy bound (10^{-6})</td>
</tr>
</tbody>
</table>
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