Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization

VIMUTH FERNANDO, University of Illinois at Urbana-Champaign, USA
KEYUR JOSHI, University of Illinois at Urbana-Champaign, USA
SASA MISAILOVIC, University of Illinois at Urbana-Champaign, USA

We present Parallely, a programming language and a system for verification of approximations in parallel message-passing programs. Parallely's language can express various software and hardware level approximations that reduce the computation and communication overheads at the cost of result accuracy.

Parallely's safety analysis can prove the absence of deadlocks in approximate computations and its type system can ensure that approximate values do not interfere with precise values. Parallely's quantitative accuracy analysis can reason about the frequency and magnitude of error. To support such analyses, Parallely presents an approximation-aware version of canonical sequentialization, a recently proposed verification technique that generates sequential programs that capture the semantics of well-structured parallel programs (i.e., ones that satisfy a symmetric nondeterminism property). To the best of our knowledge, Parallely is the first system designed to analyze parallel approximate programs.

We demonstrate the effectiveness of Parallely on eight benchmark applications from the domains of graph analytics, image processing, and numerical analysis. We also encode and study five approximation mechanisms from literature. Our implementation of Parallely automatically and efficiently proves type safety, reliability, and accuracy properties of the approximate benchmarks.

CCS Concepts: • **Computing methodologies → Parallel programming languages**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: Approximate Computing, Reliability, Accuracy, Safety

ACM Reference Format:
Vimuth Fernando, Keyur Joshi, and Sasa Misailovic. 2019. Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 119 (October 2019), 29 pages. https://doi.org/10.1145/3360545

1 INTRODUCTION

Approximation is inherent in many application domains, including machine learning, big-data analytics, multimedia processing, probabilistic inference, and sensing [Rinard 2006; Chakradhar et al. 2009; Misailovic et al. 2010; Liu et al. 2011; Goiri et al. 2015; Stanley-Marbell and Rinard 2018]. The increased volume of data and emergence of heterogeneous processing systems dictate the need to trade accuracy to reduce both *computation and communication bottlenecks*. For instance, Hogwild! has significantly improved machine learning tasks by eschewing synchronization in stochastic gradient descent computation [Recht et al. 2011], TensorFlow can reduce precision of floating-point data by transferring only some bits [Abadi et al. 2016], Hadoop can sample inputs to reductions [Goiri et al. 2015], MapReduce can drop unresponsive tasks [Dean and Ghemawat 2004]. Researchers also proposed various techniques for approximating parallel computations in software [Rinard 2007; Udupa et al. 2011; Renganarayana et al. 2012; Misailovic et al. 2013; Samadi

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART119
https://doi.org/10.1145/3360545

et al. 2014; Campanoni et al. 2015; Akram et al. 2016; Deiana et al. 2018; Khatamifard et al. 2018], and networks-on-chips [Boyapati et al. 2017; Stevens et al. 2018; Fernando et al. 2019a]. A recent survey [Betzel et al. 2018] studied over 17 different communication-related transformations such as *compression* (e.g., reducing numerical precision), *selective communication skipping* (e.g., dropping tasks or messages), *value prediction* (e.g., memoization), and *relaxed synchronization* (e.g. removing locks from shared memory).

Despite a wide variety of parallel approximations, they have been justified only empirically. Researchers designed several static analyses for verifying program approximation, but only for sequential programs. Previous works include safety analyses, such as the EnerJ type system for type safety [Sampson et al. 2011] and Relaxed RHL for relational safety [Carbin et al. 2012], analysis of quantitative reliability (the probability that the approximate computation produces *the same* result as the original) in Rely [Carbin et al. 2013b], and *accuracy* (the frequency and magnitude of error) analysis for programs running on unreliable cores in Chisel [Misailovic et al. 2014]. These prior works had stayed away from parallel programming models, in part due to the complexities involved with reasoning about arbitrary interleavings and execution changes due to transformations of the communication primitives. Providing foundations of safety and accuracy analyses for parallel programs is therefore an intriguing and challenging research problem.

1.1 Parallely Language

We present Parallely, the first approach for rigorous reasoning about the safety and accuracy of approximate parallel programs. Parallely's language supports programs consisting of distributed processes that communicate via asynchronous message-passing. Each process communicates with the others using strongly-typed *communication channels* through the common `send` and `receive` communication primitives. We identify three basic statements that are key building blocks for a variety of approximation mechanisms and include them in Parallely:

- **Conditional Send/Receive:** The `cond-send` primitive sends data only if its boolean argument is set to true. Otherwise, it informs the matching `cond-receive` primitive to stop waiting. It can be used to implement selective communication skipping transformations.
- **Probabilistic Choice:** The probabilistic choice statement $x = e_{orig}\ [p]\ e_{approx}$ will evaluate the expression from the original program (e_{orig}) with probability p, or otherwise the approximate expression (e_{approx}). It can be used to implement transformations for selectively skipping communication or computation, and value prediction.
- **Precision Conversion:** The conversion statement $x = (t')\ y$ allows reducing the precision of data that has primitive numeric types (e.g., double to float). It can be used to implement approximate data compression.

In this paper, we used these statements to represent five approximations from literature. We studied three software-level transformations that trade accuracy for performance: precision reduction, memoizing results of maps, and sampling inputs of reductions. We also studied two approximations that resume the execution after run-time errors: dropping the contribution of failed processes running on unreliable hardware, and ignoring corrupted messages transferred over noisy channels.

1.2 Verification of Safety and Accuracy

Our verification approach starts from the observation that many approximate programs implement well-structured parallelization patterns. For instance, Samadi et al. [2014] present a taxonomy of parallel patterns amenable to software-level approximation including map, partition, reduce, scan, scatter-gather, and stencil. We show that all these parallel patterns satisfy the *symmetric*

nondeterminism property – i.e., each receive statement must only have a unique matching send statement, or a set of symmetric matching send statements.

Approximation-Aware Canonical Sequentialization. Our safety and accuracy analyses rest on the recently proposed approach for *canonical sequentialization* of parallel programs [Bakst et al. 2017]. This approach statically verifies concurrency properties of asynchronous message passing programs (with simple send and receive primitives) by exploiting the symmetric nondeterminism of well-structured parallel programs. It generates a simple sequential program that over-approximates the semantics of the original parallel program. Such a *sequentialized program* exists if the original parallel program is deadlock-free. Moreover, a safety property proved on the sequentialized program also holds on the parallel program.

We present a novel version of canonical sequentialization that supports approximation statements. We prove that the safety and deadlock-freeness of the sequentialized program implies safety of the parallel approximate program. We then use this result to support the type and reliability analyses.

Type System and Relative Safety. We propose typed *approximate channels* for (1) communicating approximate data (computed by the processes) or (2) representing unreliable communication mediums [Boyapati et al. 2017; Stevens et al. 2018; Fernando et al. 2019a]. Every variable in the program can be classified as `approx` or `precise`. The design of our type system is inspired by EnerJ [Sampson et al. 2011], as we enforce that approximate data does not interfere with precise data. For instance, approximate data can be sent only through an approximate channel and received by the `receive` primitive expecting an approximate value. We prove the *type system safety* and *non-interference* properties between the approximate and precise data. The type checker operates in two steps: it first checks that each process is locally well-typed, and then checks for the agreement between the corresponding sends and receives by leveraging canonical sequentialization.

We also studied *relative safety*, another important safety property for approximate programs. It states that if an approximate program fails to satisfy some assertion, then there exists a path in the original program that would also fail this assertion [Carbin et al. 2013a]. We show that if a developer can prove relative safety of the sequentialized approximate program with respect to the sequentialized exact program, this proof will also be valid for the parallel programs.

Reliability and Accuracy Analysis. Rely [Carbin et al. 2013b] is a probabilistic analysis that verifies that a computation running on unreliable hardware produces a correct result with high probability. Its specifications are of the form $r \leq \mathcal{R}(\texttt{result})$ and mean that the exact and approximate results are the same with high probability (greater than the constant r). It has two approximation choices: arithmetic instructions that can fail and approximate memories. Chisel [Misailovic et al. 2014] extends Rely to support joint frequency and magnitude specifications. For error magnitude, it computes the absolute error intervals of variables at the end of the execution of a program with approximate operations. It adds specifications of the form $d \geq \mathcal{D}(\texttt{result})$ that mean that the deviation of the approximate result from the exact result should be at most the constant d.

We extend the reliability analysis to support the more general probabilistic choice, allowing Rely to reason about general software-level transformations in addition to the previously supported approximate hardware instructions (such as unreliable add or multiply). We prove that verifying the reliability of the sequentialized program implies the reliability of the original parallel program, given some technical conditions on the structure of the parallel programs. We do the same with Chisel's error-magnitude analysis.

1.3 Contributions

The paper makes the following contributions:

- **Language.** Parallely is a strongly-typed message-passing asynchronous language with the statements that allow implementing various program approximations.
- **Verification of Parallel Approximations.** Parallely is the first approach for verifying the safety and accuracy of approximate parallel programs using a novel approximation-aware canonical sequentialization technique.
- **Safety Analysis.** We present type analysis for parallel approximate programs and give conditions for relative safety of parallel programs.
- **Reliability and Accuracy Analysis.** We present reliability and error magnitude analyses that leverage the approximation-aware canonical sequentialization.
- **Evaluation.** We evaluate Parallely on eight kernels and eight real-world computations. These programs implement five well-known parallel communication patterns and we apply five approximations. We show that Parallely is both effective and efficient: it verifies type safety and reliability/accuracy of all kernels in under a second and all programs within 3 minutes (on average in 47.3 seconds).

2 EXAMPLE

Figure 1 presents an implementation of an image scaling algorithm. A Parallely program consists of *processes*, which execute in parallel and communicate over *typed channels*.

The program divides the image to be scaled into horizontal slices. The master process, denoted as α, sends the image to the worker processes. The left side of Figure 1 presents the code for α. We denote each worker process as β, and the set of worker processes as $Q = \{\beta_1, \beta_2, \ldots\}$. The right side of Figure 1 presents the code for β. The notation $\Pi.\beta : Q$ states that each worker process in Q shares the same code, but with β replaced by β_i inside the code for the i^{th} worker process. Each worker process scales up its assigned slice and returns it to the master process, that then constructs the complete image. To transfer the data between the processes, a developer can use the statement `send` (Line 7, process α), which should be matched with the corresponding `receive` statement (Line 4, process β).

Types of the variables in Parallely can be precise (meaning that no approximation is applied to the values) and approximate. Parallely supports integer and floating-point scalars and arrays with different precision levels (e.g., 16, 32, 64 bit). Since the channels are typed, data transfers require the

```
1  precise int[] src[10000];
2  precise int[] dst[40000];
3  precise int[] slice[4000];
4  precise int i, idx;
5
6  for β in Q do {
7    send(β, precise int[], src)
8  };
9  for β in Q do {
10   slice = receive(β, precise int[]);
11   i = 0;
12   repeat 4000 {
13     idx = β*4000+i;
14     dst[idx] = slice[i];
15     i = i+1;
16   }
17 }
```

α

$\|\ \Pi.\beta : Q$

```
1  precise int[] src[10000];
2  precise int[] slice[4000];
3
4  src = receive(α, precise int[]);
5  //scales up the assigned slice
6  slice = scaleKernel(src,β);
7  send(α, precise int[], slice);
```

β

Fig. 1. Parallely: Scale Calculation

```
precise int[] src[10000];
approx int[] dst[40000];
approx int[] slice[4000];
precise int i, idx;
approx int pass;

for β in Q do {
  send(β, precise int[], src)
};
for β in Q do {
  pass, slice = cond-receive(β, approx int[]);
  i = 0;
  repeat 4000 {
    idx = β*4000+i;
    dst[idx] = pass ? slice[i] : dst[idx-4000];
    i = i+1;
  }
}
```

α

$|| \Pi.\beta : Q$

```
precise int[] src[10000];
approx int[] slice[4000];
approx int pass;

src = receive(α, precise int[]);
//scales up the assigned slice
slice = scaleKernel(src,β);
pass = 1 [0.9999] 0;
cond-send(pass, α, approx int[], slice);
```

β

Fig. 2. Parallely: Scale with Random Task Failures

developer to specify the data type. Transferring large data structures may incur a signficant time overhead. Large arrays (e.g. the image array) are prime candidates for applying various approximate communication techniques to reduce the overhead. We assume that processes have disjoint sets of variable names and write `pid.var` to refer to variable `var` of process `pid`.

2.1 Approximate Transformation

We consider the scenario in which there exists a small chance (here 0.01%) that a Scale worker process will fail due to a hardware error. We model this source of unreliability by setting `pass` to 1 with probability 0.9999 and 0 otherwise. Figure 2 shows an approximate version of the Scale computation. The worker processes use a special version of send, `cond-send`, to send the result to the master process. `cond-send` sends an empty acknowledgement if its first argument is 0. Otherwise, it sends the message. This statement needs to be matched with a corresponding `cond-receive` statement in the master process. The `slice` array is only updated if the message is received, and the `pass` variable is also set to 1 or 0 accordingly.

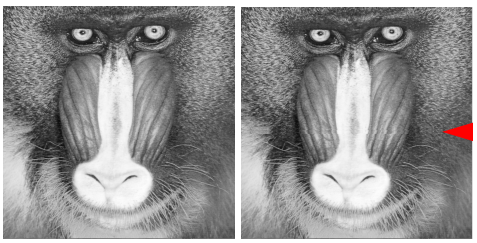


Fig. 3. Exact (left) and Approximate (right) Scaled Images

The developer may implement additional functionality to rectify the execution from such failures. While a conventional fault-recovery strategy would be re-sending the data and re-executing the failed tasks, such a strategy can incur a significant overhead of additional computation or communication. Instead, a significantly less-expensive and approximate recovery from error in this domain would be to reuse the previously computed pixels from the adjacent image positions. In the master process from Figure 2, we implemented a simple version of this strategy. If the master process receives the data (`pass` is `true`), it copies the received `slice` into `dest`. If it does not receive the data (`pass` is `false`), the previous `slice` is duplicated.

We developed a translator for Parallely, which does source-to-source translation to the Go language. Figure 3 shows the images produced by the exact version (left) and approximate version (right). The red triangle on the right side indicates the region in which the pixels have been approximated. The peak-signal-to-noise ratio is 38.8dB, indicating an acceptable approximation.

2.2 Properties

We wish to verify the following properties about this program:

- **Type Safety:** approximate variables cannot affect the values of precise variables, either directly, via assignment, or indirectly, by affecting control flow;
- **Deadlock-Freeness:** the execution of the approximate program is deadlock-free; and
- **Quantitative Reliability:** the result will be calculated correctly with probability at least 0.99. We encode this requirement in Parallely as $0.99 \leq \mathcal{R}(\text{dst})$, using the notation from Rely.

Next, we show how Parallely's static analyses verify these properties.

2.3 Verification

Parallely verifies that approximate variables do not interfere with precise variables via a type checking pass in two steps. In the first step, it checks that the code in the master process and the worker process has the correct type annotations, i.e., an approximate value cannot be assigned to the precise variable. In the second step, it uses program sequentialization (which is sensitive to the types in the send and receive primitives) to ensure that precise sends are consumed by precise receives.

```
1  precise int[] α.src[10000];
2  approx int[] α.dst[40000];
3  approx int[] α.slice[4000];
4  precise int α.i, α.idx;
5  approx int α.pass;
6  precise int[] β1.src[10000], β2.src[10000], ...;
7  approx int[] β1.slice[4000], β2.slice[4000], ...;
8  approx int β1.pass, β2.pass, ...;
9
10 for β in Q do {
11   β.src = α.src;
12 };
13 for β in Q do {
14   //scales up the assigned slice
15   β.slice = scaleKernel(β.src,β);
16   β.pass = 1 [0.9999] 0;
17   α.pass = β.pass ? 1 : 0;
18   α.slice = β.pass ? β.slice : α.slice;
19   α.i = 0;
20   repeat 4000 {
21     α.idx = β*4000+α.i;
22     α.dst[α.idx] = α.pass ? α.slice[α.i]
23                           : α.dst[α.idx-4000];
24     α.i = α.i+1;
25   };
26 }
```

α

Fig. 4. Parallely: Scale Sequential Code.

To ensure that the approximate and precise channels are matched, and to prove that the program is deadlock free, Parallely converts the program to an equivalent sequential program, shown in Figure 4. It does so by matching each `send` or cond-send with the corresponding `receive` or cond-receive and converting the message transmission operation to an assignment operation. This conversion is achieved by applying a selection of rewrite rules that perform syntactic transformations on statements in the parallel program. One rewrite rule replaces each `send` statement with a skip statement (no operation) and stores the value being sent in the context of the rewrite system. A second rewrite rule replaces the matching `receive` statement with an assignment, where the assigned value is the value that was sent by the matching `send` statement. Line 11 and line 18 in Figure 4 show how the rewrite rules sequentialize the communication from our example into assignments. Successful sequentialization guarantees that there are no deadlocks in the program *and* that the precise sends (resp. approximate sends) are matched with precise receives (resp. approximate receives).

Finally, Parallely performs a reliability analysis pass on the sequentialized program from Figure 4 to verify that the reliability of the `dest` array at the end of execution is at least 0.99. The sequential Rely analysis requires a finite bound on the number of loop iterations. Here, it is the number of the worker processes, $|Q|$. Our reliability analysis on the sequentialized program results in the constraint: $0.9999^{|Q|} \geq 0.99$. The formula is satisfied for every $|Q| \leq 100$. Our paper shows that if the reliability predicate is valid for the sequentialized program, it will also be valid for the parallel program.

n	$\in \mathbb{N}$	*quantities*		
m	$\in \mathbb{N} \cup \mathbb{F} \cup \{\varnothing\}$	*values*	$S \rightarrow$	
r	$\in [0, 1.0]$	*probability*	skip	*empty program*
x, b, X	$\in$ Var	*variables*	$\mid x$ = *Exp*	*assignment*
a	$\in$ ArrVar	*array variables*	$\mid x$ = *Exp* $[r]$ *Exp*	*probabilistic choice*
α, β	$\in$ Pid	*process ids*	$\mid x = b?$ *Exp* : *Exp*	*conditional choice*
			$\mid S ; S$	*sequence*
Exp	$\rightarrow m \mid x \mid f(Exp^*) \mid$	*expressions*	$\mid x = a[Exp^+]$	*array load*
	$(Exp) \mid Exp\ op\ Exp$		$\mid a[Exp^+] = Exp$	*array store*
			$\mid$ if $x\ S\ S$	*branching*
q	$\rightarrow$ precise | approx	*type qualifiers*	$\mid$ repeat n {S}	*repeat n times*
t	$\rightarrow$ int<n> | float<n>	*basic types*	$\mid x$ = (T)*Exp*	*cast*
T	$\rightarrow q\ t \mid q\ t\ []$	*types*	$\mid$ for $i : [Pid^+]\{S\}$	*iterate over processes*
D	$\rightarrow$ T x | T a$[n^+]$ |	*variable*	$\mid$ send(α, T, x)	*send message*
	$D ; D$	*declarations*	$\mid x$ = receive(α, T)	*receive a message*
			$\mid$ cond-send(b, α, T, x)	*conditionally send*
P	$\rightarrow [D; S]_\alpha$ |	*process*	$\mid b, x$ = cond-receive(α, T)	*receive from a*
	$\Pi.\alpha : X\ [D; S]_\alpha \mid$	*process group*		*cond-send*
	$P \parallel P$	*process composition*		

Fig. 5. Parallely Syntax

3 VERIFYING SAFETY AND ACCURACY OF TRANSFORMATIONS

Figure 5 presents the Parallely syntax. Parallely is a strongly typed imperative language with primitives for asynchronous communication. The send statement is used to asynchronously send a value to another process using a unique process identifier. The receiving process can use the blocking `receive` statements to read the message. In addition, Parallely supports array accesses, iteration over a set of processes, conditionals, and precision manipulation via casting. We present the precise semantics of the Parallely language in Section 4.

Given an approximate version (P^A) of a program (P), Parallely first type checks each individual process using the rules defined in Section 6. Then, it converts the approximate program to its canonical sequentialization (P^A_{seq}) using the procedure described in Section 5.1, which proves deadlock freedom. Finally Parallely performs the reliability and accuracy analysis on the sequentialized program (Section 7).

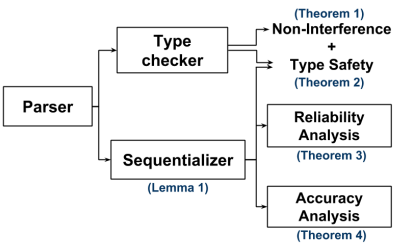


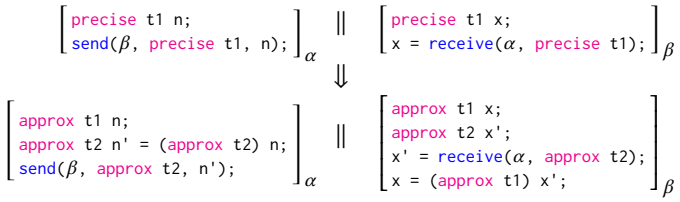
Fig. 6. Overview of Parallely

Figure 6 presents the overview of the modules in our implementation of Parallely. The type checker and sequentializer modules work together to provide the safety guarantees. The sequentialization module outputs a sequential program, which can then be used with the reliability/accuracy analysis. Figure 6 also highlights the relevant lemma or theorem in this paper for each aspect.

We next present several popular parallel patterns and approximate transformations from literature, all of which can be represented and verified using Parallely. For each pattern, we present a code example, transformation, and discuss verification challenges. Full details, including the sequentialized code are given in Appendix D [Fernando et al. 2019b]. We also present the analysis time for these patterns in Section 8.

3.1 Precision Reduction

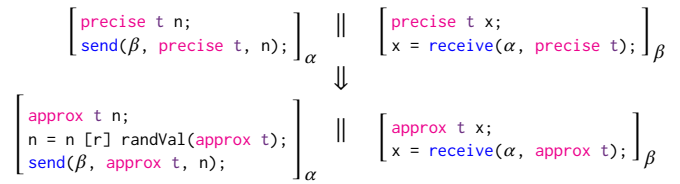
Pattern and Transformation: It reduces the precision of approximate data being transferred between processes by converting the original data type to a less precise data type (e.g. doubles to floats). Precision reduction is a common technique for approximate data compression (e.g. as used in TensorFlow [Abadi et al. 2016]).



Safety: As precision reduction is an approximate operation, the type of the reduced-precision data must be an approximate type. The type must be changed in both the sender and receiver process to the same type. Further, there must not already be messages of the less precise type being sent between the processes, else the converted code may affect the order of the messages and violate the symmetric nondeterminism property necessary for sequentialization.
Accuracy: The developer specifies the domain of the transmitted value as an interval (e.g., $[0, 32]$). Precision reduction introduces an error to this transmitted value that depends on the original and converted types (e.g., 10^{-19} when converting from double to float). The interval analysis in Section 7 can then calculate the maximum absolute error of the result.

3.2 Data Transfers over Noisy Channels

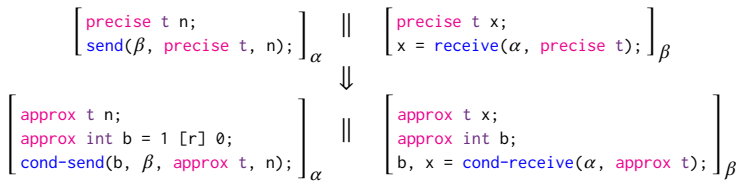
Pattern and Transformation: It models the transfer of approximate data over an unreliable communication channel. The channel may corrupt the data and the receiver may receive a garbage value with probability $1 - r$. If the received message is corrupted, the approximate program may still decide to continue execution (instead of requesting resend).



Safety: The variable that may potentially be corrupted must have an approximate type. Consequently, the developer must send and receive the data over an approximate typed channel.
Accuracy: We use a reliability specification $r \leq \mathcal{R}(\beta.x)$ which states that the variable being transferred (x) must reach the destination intact with probability at least r. This specification can be directly proved on the sequentialized version of the program, with a single statement (the probabilistic choice modeling the occasional data corruption) affecting this reliability condition.

3.3 Failing Tasks

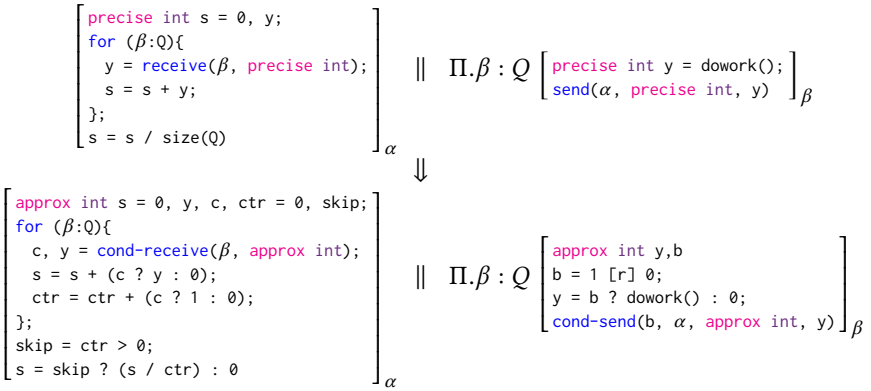
Pattern and Transformation: It models the execution of tasks that can fail with some probability $1 - r$ due to hardware or software errors. For instance, MapReduce ignores a task result if the task experiences a failure [Dean and Ghemawat 2004]; Topaz returns an error if a task running on an unreliable core fails [Achour and Rinard 2015]. We model such scenarios by conditionally transferring data (with cond-send and cond-receive), based on the random chance of task success, r.



Safety: Like in the noisy-channel pattern, the developer needs to assign approximate types to the data being sent and the channels over which the data is sent. In addition, the developer must use cond-send and cond-receive calls.
Accuracy: Similar to the noisy-channel pattern, we again use a Rely style specification, e.g. $r \leq \mathcal{R}(\beta.x)$, which can be proved on the sequentialized version of the program.

3.4 Approximate Reduce (Sampling)

Pattern and Transformation: This pattern approximates an aggregation operation such as finding the maximum or sum. To implement sampling, the worker process only computes and sends the result with probability r. Otherwise, it only sends an empty message to the master. The master process adjusts the aggregate based on the number of received results (but only if it received *some* data).



Safety: To successfully sequentialize, the transformed program's master task gathers the results from all symmetric workers tasks. To ensure that divide-by-zero cannot happen, we need an additional check for `ctr`.
Accuracy: We can automatically prove two properties: the reliability (as before), and the interval bound of the result. If the inputs are in the range $[a, b]$, then the error of the average will also be in the same range. Further formal reasoning about this pattern (e.g., [Zhu et al. 2012]) may provide more interesting probabilistic bounds. However, application of such analyses is outside of the scope of this paper.

3.5 Approximate Map (Approximate Memoization)

Pattern and Transformation: A map task computes on a list of independent elements. Reduction of the number of tasks in conjunction with approximate memoization [Chaudhuri et al. 2011; Samadi et al. 2014] can reduce communication and improve energy efficiency. If the master process decides not to send a task to a worker process, then that worker process will return an empty result. Upon receiving an empty result, the master process uses the previously received result in its place. There

is no need for additional code to use the most recently received result, as cond-receive does not update the variable that stores the received value when an empty value is received.

The example in Figure 2 uses this pattern. If a task fails to return a slice to the master task then the previous slice is used, as described in Section 2.

Safety: Even if no work is sent to some workers, the master task must still receive an empty message from each worker to ensure symmetric nondeterminism.

Accuracy: We use a reliability specification $r \leq \mathcal{R}(\alpha.results)$ which states that the reliability of the entire `results` array is at least r. If even one element of the array is different from the precise version, then the entire array is considered incorrect for the purposes of measuring reliability. This reliability depends on the probability of sending a job to a worker task in the master task.

3.6 Other Verified Patterns and Transformations

Multiple other computation patterns can be expressed in Parallely in a manner that satisfies symmetric nondeterminism. These include the Scatter-Gather, Stencil, Scan, and Partition patterns. These patterns are similar to the map and reduce patterns, but distribute data slightly differently to the worker tasks. We can apply transformations such as precision reduction, failing tasks, sampling, etc. to these patterns and prove their sequentializability and type safety. We can also calculate their reliability and accuracy. We give the full details in Appendix D.

We also support Rely's approximate instructions and approximate memories. For example, we can model arithmetic instructions as `: z = (x op y) [r] randVal()`, which models an instruction that can produce an error with probability `r`. Similarly, approximate memories can be modeled through the corresponding read and write operations, e.g., as `x = x [r] randVal()`, which corrupts the memory location storing a variable `x` with probability `r`.

3.7 Unsafe Patterns and Transformations

Runtime Task Skipping. Certain approximations are not safe, as they can introduce deadlocks or violate relative safety properties. For example, if the approximate reduce transformation is implemented by simply not sending some data back from the workers, it may cause the master process to wait for data that it will never receive, violating the symmetry requirement necessary for sequentialization.

Timed Receives. Another possible type of approximate receive operation is the timed receive operation, which times out if no value is received within a specified time bound. Such timed receives will not work with our approach, as they introduce the possibility of sending a value that is not received. However, we anticipate that recent approaches like [Gleissenthall et al. 2019] (which support this type of timed communication using some simplifying assumptions), could extend the reach of our analysis to support timed receive operations.

Iterative Fixed-Point Computations. A common computation pattern is to repeat a calculation until the errors are small. This pattern does not satisfy the property of non-interference in Parallely even though it is a safe computation. In addition, if there is communication within the loop body, the loop cannot be sequentialized, as it uses the loop carried state for termination. Sequentialization requires that the decision only depends on values computed in the current iteration.

$$\left[\begin{array}{l}\texttt{approx float error,oldresult;}\\ \texttt{//...}\\ \texttt{while(error > 0.1)}\\ \quad\texttt{approx float result = loop\_body()}\\ \quad\texttt{error = abs(oldresult-result);}\\ \quad\texttt{oldresult = result;}\end{array}\right]_{\alpha} \;||\; \Pi.\beta : Q \left[\ \ldots\ \right]_{\beta}$$

$$\text{E-Var-C}\ \frac{\langle n_b, \langle 1 \rangle\rangle = \sigma(x)}{\langle x, \sigma, h\rangle \rightarrow^{1}_{\psi} \langle h(n_b), \sigma, h\rangle} \qquad \text{E-Iop-R1}\ \frac{\langle e_1, \sigma, h\rangle \rightarrow^{1}_{\psi} \langle e_1', \sigma, h\rangle}{\langle e_1\ op\ e_2, \sigma, h\rangle \rightarrow^{1}_{\psi} \langle e_1'\ op\ e_2, \sigma, h\rangle}$$

$$\text{E-Iop-R2}\ \frac{\langle e_2, \sigma, h\rangle \rightarrow^{1}_{\psi} \langle e_2', \sigma, h\rangle}{\langle n\ op\ e_2, \sigma, h\rangle \rightarrow^{1}_{\psi} \langle n\ op\ e_2', \sigma, h\rangle} \qquad \text{E-Iop-C}\ \frac{}{\langle n_1\ op\ n_2, \sigma, h\rangle \rightarrow^{1}_{\psi} \langle op(n_1, n_2), \sigma, h\rangle}$$

Fig. 7. Dynamic Semantics of Expressions (Selection)

4 SEMANTICS OF PARALLELY

Figure 7 and Figure 8 present the Parallely's most important rules for the small-step expression and statement semantics. The remaining (standard) rules are presented in the Appendix A [Fernando et al. 2019b].

References. A *reference* is a pair $\langle n_b, \langle n_1, \ldots, n_k\rangle\rangle \in \texttt{Ref}$ that consists of a base address $n_b \in Loc$ and a dimension descriptor $\langle n_1, \ldots, n_k\rangle$. References describe the location and the dimension of variables in the heap.

Frames, Stacks, and Heaps. A *frame* σ is an element of the domain $\mathrm{E} = \mathrm{Var} \rightarrow \mathrm{Ref}$ which is the set of finite maps from program variables to references. A *heap* $h \in H = \mathbb{N} \rightarrow \mathbb{N} \cup \mathbb{F} \cup \{\varnothing\}$ is a finite map from addresses (integers) to values. Values can be an Integer, Float or the special *empty message* ($\varnothing$).

Processes. Individual processes execute their statements in sequential order. Each process has a unique process identifier (Pid). Processes can refer to each other using the process identifier. We do not discuss process creation and removal. We assume that the processes have disjoint variable sets of variable names. We write `pid.var` to refer to variable `var` of process `pid`. When unambiguous, we will omit `pid` and just write `var`.

Types. Types in Parallely are either precise (meaning that no approximation can be applied to them) and approximate. Parallely supports integer and floating-point scalars and arrays with different levels of precision.

Typed Channels and Message Orders. Processes communicate by sending and receiving messages over a typed *channel*. There is a separate subchannel for each pair of processes further split by the *type* of message. $\mu \in Channel = Pid \times Pid \times Type \rightarrow Val^*$. Messages on the same subchannel are delivered in order but there are no guarantees for messages sent on separate (sub)channels.

Programs. We define a program as a parallel composition of processes. We denote a program as $P = [P]_1 \parallel \cdots \parallel [P]_i \parallel \cdots \parallel [P]_n$. Where $1, \ldots, n$ are process identifiers. An approximated program executes within *approximation model*, ψ, which in general may contain the parameters for approximation (e.g., probability of selecting original or approximate expression). We define special reliable model 1_ψ, which evaluates the program without approximations.

Global and Local Environments. Each process works on its private environment consisting of a frame and a heap, $\langle \sigma^i, h^i\rangle \in \Lambda = H \times \mathrm{E}$. We define a global configuration as a triple $\langle P, \epsilon, \mu\rangle$ of a program, global environment, and a channel. The global environment is a map from the process identifiers to the local environment $\epsilon \in Env = Pid \mapsto \Lambda$.

Expressions. Figure 7 presents the dynamic semantics for expressions. The labeled small-step evaluation relation of the form $\boxed{\langle e, \sigma, h\rangle \rightarrow^{1}_{\psi} \langle e', \sigma, h\rangle}$ states that from a frame σ and a heap h, an expression e evaluates in one step with probability 1 to an expression e' without any changes to

$$\textsc{E-Assign-R}\ \frac{\langle e, \sigma, h\rangle \rightarrow^{1}_{\psi} \langle e', \sigma, h\rangle}{\langle x = e, \sigma, h, \mu\rangle \rightarrow^{1}_{\psi} \langle x = e', \sigma, h, \mu\rangle} \qquad \textsc{E-Assign-C}\ \frac{\langle n_b, \langle 1\rangle\rangle = \sigma(x)}{\langle x = n, \sigma, h, \mu\rangle \rightarrow^{1}_{\psi} \langle \texttt{skip}, \sigma, h[n_b \mapsto n], \mu\rangle}$$

$$\textsc{E-Assign-Prob-True}\ \frac{}{\langle x = e_1\ [r]\ e_2, \sigma, h, \mu\rangle \rightarrow^{r}_{\psi} \langle x = e_1, \sigma, h, \mu\rangle} \qquad \textsc{E-Assign-Prob-False}\ \frac{}{\langle x = e_1\ [r]\ e_2, \sigma, h, \mu\rangle \rightarrow^{1-r}_{\psi} \langle x = e_2, \sigma, h, \mu\rangle}$$

$$\textsc{E-Par-Iter}\ \frac{}{\langle \texttt{for}\ i : [\alpha_1, \ldots, \alpha_k]\{S\}, \sigma, h, \mu\rangle \rightarrow^{1}_{\psi} \langle S[\alpha_1/i]; \ldots; S[\alpha_k/i], \sigma, h, \mu\rangle}$$

$$\textsc{E-Send}\ \frac{\text{isPid}(\beta) \qquad \langle n_b, \langle 1\rangle\rangle = \sigma(y) \qquad h[n_b] = n \qquad \mu[\langle \alpha, \beta, t\rangle] = m}{\langle [\text{send}(\beta, t, y)]_\alpha, \sigma, h, \mu\rangle \rightarrow^{1}_{\psi} \langle \texttt{skip}, \sigma, h, \mu[\langle \alpha, \beta, t\rangle \mapsto m + +n]\rangle}$$

$$\textsc{E-Receive}\ \frac{\mu[\langle \beta, \alpha, t\rangle] = m :: n \qquad \langle n_b, \langle 1\rangle\rangle = \sigma(x) \qquad \text{isPid}(\beta)}{\langle [x = \text{receive}(\beta, t)]_\alpha, \sigma, h, \mu\rangle \rightarrow^{1}_{\psi} \langle \texttt{skip}, \sigma, h[n_b \mapsto m], \mu[\langle \beta, \alpha, t\rangle \mapsto n]\rangle}$$

$$\textsc{E-CondSend-True}\ \frac{\begin{array}{c}\langle l, \langle 1\rangle\rangle = \sigma(b) \qquad h[l] \neq 0 \qquad \text{isPid}(\beta) \\ \langle n_b, \langle 1\rangle\rangle = \sigma(y) \qquad h[n_b] = v \qquad \mu[\langle \alpha, \beta, t\rangle] = m\end{array}}{\langle [\text{cond-send}(b, \beta, t, y)]_\alpha, \sigma, h, \mu\rangle \rightarrow^{1}_{\psi} \langle \texttt{skip}, \sigma, h, \mu[\langle \alpha, \beta, t\rangle \mapsto m + +n]\rangle}$$

$$\textsc{E-CondSend-False}\ \frac{\begin{array}{c}\langle l, \langle 1\rangle\rangle = \sigma(b) \qquad h[l] = 0 \qquad \text{isPid}(\beta) \\ \mu[\langle \alpha, \beta, t\rangle] = m \qquad \mu' = \mu[\langle \alpha, \beta, t\rangle \mapsto m + +\varnothing]\end{array}}{\langle [\text{cond-send}(b, \beta, t, y)]_\alpha, \sigma, h, \mu\rangle \rightarrow^{1}_{\psi} \langle \texttt{skip}, \sigma, h, \mu'\rangle}$$

$$\textsc{E-CondReceive-True}\ \frac{\begin{array}{c}\mu[\langle \beta, \alpha, t\rangle] = m :: n \qquad m \neq \varnothing \qquad \langle n_1, \langle 1\rangle\rangle = \sigma(x) \qquad \langle n_2, \langle 1\rangle\rangle = \sigma(b) \\ h' = h[n_1 \mapsto m][n_2 \mapsto 1] \qquad \mu' = \mu[\langle \beta, \alpha, t\rangle \mapsto n]\end{array}}{\langle [b, x = \text{cond-receive}(\beta, t)]_\alpha, \sigma, h, \mu\rangle \rightarrow^{1}_{\psi} \langle \texttt{skip}, \sigma, h', \mu'\rangle}$$

$$\textsc{E-CondReceive-False}\ \frac{\begin{array}{c}\mu[\langle \beta, \alpha, t\rangle] = \varnothing :: m \qquad \langle n_b, \langle 1\rangle\rangle = \sigma(b) \\ h' = h[n_b \mapsto 0] \qquad \mu' = \mu[\langle \beta, \alpha, t\rangle \mapsto m]\end{array}}{\langle [b, x = \text{cond-receive}(\beta, t)]_\alpha, \sigma, h, \mu\rangle \rightarrow^{1}_{\psi} \langle \texttt{skip}, \sigma, h', \mu'\rangle}$$

Fig. 8. Process-Level Dynamic Semantics of Statements (Selection)

$$\textsc{E-Assign-Prob-Exact}\ \frac{}{\langle x = e_1\ [r]\ e_2, \sigma, h, \mu\rangle \rightarrow^{1}_{1_\psi} \langle x = e_1, \sigma, h, \mu\rangle} \qquad \textsc{E-Cast-Exact}\ \frac{}{\langle x = (\text{T})e, \sigma, h, \mu\rangle \rightarrow^{1}_{1_\psi} \langle x = e, \sigma, h, \mu\rangle}$$

Fig. 9. Exact Execution Semantics of Statements (Selection)

$$\textsc{GLOBAL-STEP}\ \frac{p_\alpha = P_s[\alpha \mid (\epsilon, \mu, P_\alpha \parallel P_\beta)] \qquad \epsilon[\alpha] = \langle \sigma, h\rangle \qquad \langle P_\alpha,\ \sigma, h, \mu\rangle \rightarrow^{p}_{\psi} \langle P'_\alpha,\ \sigma', h', \mu'\rangle \qquad p' = p \cdot p_\alpha}{(\epsilon, \mu, P_\alpha \parallel P_\beta) \xrightarrow{\alpha, p'}_{\psi} (\epsilon[\alpha \mapsto \langle \sigma', h'\rangle], \mu', P'_\alpha \parallel P_\beta)}$$

Fig. 10. Global Dynamic Semantics

the frame σ and heap h. Parallely supports typical integer and floating point operations. We allow function calls and inline them as a preliminary step.

Statements. Figure 8 defines the semantics for statements. The small-step relation of the form $\boxed{\langle s, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle s', \sigma', h', \mu'\rangle}$ defines a single process in the program evaluating in its local frame σ, heap h, and the global channel μ. Individual processes can only access their own frame and heap. We unroll `repeat` statements as a preliminary step. We use `h::t` to denote accessing the head (`h`) of a queue and `h++t` to denote adding `t` to the end of the queue.

Approximate Statements. In addition to the usual statements, we include probabilistic choice and Boolean choice. A probabilistic choice expression $x = e_1\ [r]\ e_2$ evaluates to e_1 with probability r (e_2 with $1 - r$) if r is float, or as a deterministic if-expression when r is an integer, selecting e_1 if $r \geq 1$ (e_2 if $r = 0$). We use the *cast* statement to perform precision reduction (but only between the values of the same general type, int or float).

Parallely also contains `cond-send` statements that use an additional *condition variable* and only sends the message if it evaluates to 1. If the message is not sent, an empty message ($\varnothing$) is sent to the channel as a signal. `cond-receive` acts similar to `receive` but only updates the variables if the received value is not $\varnothing$.

In Parallely non-deterministic differences in evaluations arise only from the probabilistic choice expressions. These can be used to model a wide range of approximate transformations. We use the approximation model to differentiate between exact and approximate executions of the program. We use 1_{ψ} to specify exact, precise execution and present program semantics of exact execution in Figure 9. Under exact execution, probabilistic choice statements always evaluate to the first option, casting performs no change, and all declarations allocate the memory required to store the full precision data.

For reliability analysis to be valid we require that the approximate program under exact evaluation be the same as the program without any approximations.

Global Semantics. Figure 10 defines the global semantics. Small step transitions of the form $\boxed{(\epsilon, \mu, P_\alpha \parallel P_\beta) \xrightarrow{\alpha, p}_{\psi} (\epsilon', \mu', P'_\alpha \parallel P_\beta)}$ define a single process α taking a local step with probability p. The distribution $P_s(i \mid \langle P, \epsilon, \mu\rangle)$ models the probability that the process with id i is scheduled next. We define it history-less and independent of ϵ contents. For reliability analysis we assume a fair scheduler that in each step has a positive probability for all threads that can take a step in the program. The global semantics consists only of individual processes executing using the statement semantics in their local environment and the shared μ.

4.1 Big-Step Notations

Since we are concerned only with the halting states of processes and analysis of deadlock free programs, we will define big-step semantics as follows for parallel traces that begin and end with an empty channel:

Definition 1 (Trace Semantics for Parallel Programs).

$$\langle \cdot,\ \epsilon\rangle \overset{\tau,\, p}{\Longrightarrow}_{\psi} \epsilon' \equiv \langle \epsilon,\ \varnothing,\ \cdot\rangle \xrightarrow{\lambda_1,\, p_1}_{\psi} \ \ldots\ \xrightarrow{\lambda_n,\, p_n}_{\psi} \langle \epsilon',\ \varnothing,\ \texttt{skip}\rangle$$
$$\textit{where } \tau = \lambda_1, \ldots, \lambda_n, p = \prod_{i=1}^{n} p_i$$

This big-step semantics is the reflexive transitive closure of the small-step global semantics for programs and records a *trace* of the program. A trace $\tau \in T \to \cdot | \alpha :: T$ is a sequence of small step global transitions. The probability of the trace is the product of the probabilities of each transition.

$$\textsc{R-Send}\ \frac{\Delta \models x = \beta \qquad \beta \text{ is a Pid} \qquad \Gamma[\alpha, \beta, t] = m \qquad \Gamma' = \Gamma[\alpha, \beta, t \mapsto m + +y]}{\Gamma, \Delta, [\text{send}(x, t, y)]_\alpha \rightsquigarrow \Gamma', \Delta, \texttt{skip}}$$

$$\textsc{R-Receive}\ \frac{\Delta \models x = \beta \qquad \beta \text{ is a Pid} \qquad \Gamma[\beta, \alpha, t] = m :: n \qquad \Gamma' = \Gamma[\beta, \alpha, t \mapsto n] \qquad \Delta' = \Delta; \text{y} = \text{m}}{\Gamma, \Delta, [y = \text{receive}(x, t)]_\alpha \rightsquigarrow \Gamma', \Delta', \texttt{skip}}$$

$$\textsc{R-CondSend}\ \frac{\Delta \models x = \beta \qquad \beta \text{ is a Pid} \qquad \Gamma[\alpha, \beta, t] = m \qquad \Gamma' = \Gamma[\alpha, \beta, t \mapsto m + +(b : y)]}{\Gamma, \Delta, [\text{cond-send}(b, x, t, y)]_\alpha \rightsquigarrow \Gamma', \Delta, \texttt{skip}}$$

$$\textsc{R-CondReceive}\ \frac{\Delta \models x = \beta \qquad \beta \text{ is a Pid} \qquad \Gamma[\beta, \alpha, t] = (b' : m) :: n \qquad \Gamma' = \Gamma[\beta, \alpha, t \mapsto n] \qquad \Delta' = \Delta; \text{b} = b' ? \ 1 : 0; \text{y} = b' ? \ m : x}{\Gamma, \Delta, [b, y = \text{cond-receive}(x, t)]_\alpha \rightsquigarrow \Gamma', \Delta', \texttt{skip}}$$

$$\textsc{R-Context}\ \frac{\Gamma, \Delta, A \rightsquigarrow \Gamma', \Delta', A'}{\Gamma, \Delta, \text{A}; \text{B} \rightsquigarrow \Gamma', \Delta', \text{A'}; \text{B}}$$

Fig. 11. A Selection of the Rewrite Rules

Definition 2 (Aggregate Semantics for Parallel Programs).

$$\langle \cdot,\ \epsilon \rangle \overset{p}{\Longrightarrow}_\psi \epsilon' \text{ where } p = \sum_{\tau \in \text{T}} p_\tau \text{ such that } \langle \cdot,\ \epsilon \rangle \overset{\tau, p_\tau}{\Longrightarrow}_\psi \epsilon'$$

The big-step aggregate semantics enumerates over the set of all finite length traces and sums the aggregate probability that a program starts in an environment ϵ and terminates in an environment ϵ'. It accumulates the probability over all possible interleavings that end up in the same final state.

Termination and Errors. Halted processes are those processes that have finished executing permanently. A process α is a halted process, i.e. $\alpha \in \text{hprocs}(\epsilon, \mu, P)$ if any of the following hold: (1) α's remaining program is `skip` (correct execution), (2) α is stuck waiting for a message, when its next statement is a receive or cond-receive, but there is no matching send or cond-send in the rest of the program (error state).

5 APPROXIMATION-AWARE CANONICAL SEQUENTIALIZATION

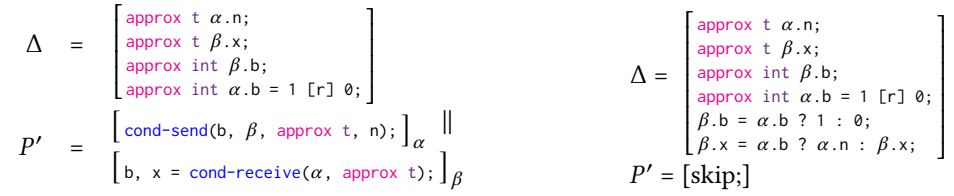
Canonical sequentialization by Bakst et al. [2017] is a method for statically verifying concurrency properties of asynchronous message passing programs. It leverages the structure of the parallel program to derive a representative sequential execution, called a *canonical sequentialization*. Verifying the properties on this sequential version would imply their validity in parallel execution too.

One major requirement for sequentialization is that the parallel program must be *symmetrically nondeterministic* – each receive statement must only have a unique matching send statement, or a set of symmetric matching send statements. Further, there must not be spurious send statements that do not have a matching receive statement. The procedure for checking that a program is symmetrically nondeterministic is discussed by Bakst et al. [2017, Section 5].

5.1 Sequentialization of Parallely Programs

We define rewrite rules of the form $\boxed{\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta', P'}$ which consist of a context (Γ), sequential prefix (Δ), and the remaining program to be rewritten (P). The prefix Δ contains the part of the program that has already been sequentialized. The context Γ consists of a symbolic set of messages in flight – *variables* being sent, but their matching receive has not already been found and sequentialized, and assertions about process identifiers.

A selection of the rewriting rules are available in Figure 11. They aim to fully sequentialize the program, i.e., reach $(\varnothing, \Delta_{prog}, \texttt{skip})$. The rules aim to gradually replace statements from the parallel program with statements in the sequential program Δ_{prog}. The sequential program is equivalent to the parallel program (as further discussed in Lemma 1). We define $\rightsquigarrow^*$ to be the transitive closure of rewrite rules. The sequentialization process starts from an empty context and sequential prefix,



(a) An intermediate step in the rewriting process

(b) Final sequentialized code

Fig. 12. An Example of the Rewriting Process.

along with the original program, and applies the rewrite steps until the program is rewritten to `skip` with a context having empty message buffers.

Preliminaries. To support the rewrite process in our imperative language and avoid side effects we ensure that only variables can appear in `send` statements and that the program is in a Single Static Assignment (SSA) form before the rewriting process. This ensures that our rewrite context correctly represents the state of the variables that are being communicated. We also provide syntactic sugar to represent processing sending entire arrays. We de-sugar such statements to be a set of sequential `send` statements for each array location. In addition, we rename all variables in the program to ensure that the variable sets used in individual processes are disjoint. We place some restrictions on Parallely programs to simplify the rewriting process: we do not allow communication with external processes and do not allow communication inside conditional statements.

Guarded Expressions. In addition to the available rewrite rules in Bakst et al. [2017] we define *guarded expressions* to support our rewriting steps for `cond-send` and `cond-receive` statements. As described in Section 4, `cond-send` statements only add a message to a channel if a *guard* (b) evaluates to 1. To represent this effect in the rewrite context instead of adding the sent variable to the context, we use a guarded expression.

We extend the interpretation of contexts in an environment ϵ, written as $[\![\Gamma(\alpha, \beta, t)]\!]_\epsilon$ from Bakst et al. [2017] as follows for guarded expressions:

$$\text{If } \Gamma(\alpha, \beta, t) = x : b,\ [\![\Gamma(\alpha, \beta, t)]\!]_\epsilon = \begin{cases} \epsilon_\alpha[x] & \text{if } \epsilon_\alpha[b] = 1 \\ \varnothing, & \text{else} \end{cases}$$

where $\epsilon_\alpha[b]$ is the value of the variable b in the process α. Guarded expressions are used during the sequentialization of conditional sends and receives.

Example. Figure 12 illustrates the sequentialization process for the example program from Section 3.3 that models the execution of tasks that can fail with some probability. We reach the intermediate step in Figure 12(a) by applying the R-Context rule multiple times on statements that do not perform communication. Next, we apply the R-CondSend rule, which is the only applicable rule in this step. This rule saves the message being sent as a guarded expression $n : b$ in the context (i.e. $\Gamma(\alpha, \beta, \texttt{approx t}) = n : b$). Finally, we apply the R-CondReceive rule, which retrieves the guarded expression from the context and assigns to the variables in the receiver process. The final sequentialized program is shown in Figure 12(b).

Rewrite Soundness. Programs are in a *normal form* if they are parallel compositions of statements from distinct processes, i.e. statements from the same process are not composed in parallel. For two programs P_1 and P_2 in normal form, we define $P_1 \circ P_2$ as the process-wise sequencing of P_2 after P_1, i.e., for each process p present in both P_1 and P_2, the statements for p in P_1 are executed before those in P_2. We define $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]$ to indicate that ϵ and μ are a store and message buffer consistent with states reachable by executing Δ and assumptions in Γ. Let $\epsilon|_P$ denote the store

$$\textsc{TR-Prob}\ \frac{\Theta \vdash e_1 : q\ t \qquad \Theta \vdash e_2 : q'\ t \qquad \Theta \vdash x : \texttt{approx}\ t}{\Theta \vdash \texttt{x} = e_1\ [r]\ e_2 : \Theta}$$

$$\textsc{TR-CondChoice}\ \frac{\Theta \vdash e_1 : q\ t \qquad \Theta \vdash e_2 : q'\ t \qquad \Theta \vdash x : \texttt{approx}\ t \qquad \Theta \vdash b : q''\ \texttt{int}}{\Theta \vdash \texttt{x} = b?\ e_1 : e_2 : \Theta}$$

$$\textsc{TR-Cast}\ \frac{\Theta \vdash x : \texttt{approx}\ t \qquad \Theta \vdash e : q\ t'}{\Theta \vdash \texttt{x} = (\texttt{approx}\ \texttt{t})\ \texttt{e} : \Theta}$$

$$\textsc{TR-CondSend}\ \frac{\Theta \vdash b : q\ \texttt{int} \qquad \Theta \vdash y : \texttt{approx}\ t \qquad T = \texttt{approx}\ t}{\Theta \vdash \text{cond-send}(b,\ \beta,\ T,\ y) : \Theta}$$

$$\textsc{TR-CondReceive}\ \frac{\Theta \vdash x : \texttt{approx}\ t \qquad T = \texttt{approx}\ t \qquad \Theta \vdash b : \texttt{approx int}}{\Theta \vdash b,\ x = \text{cond-receive}(\beta,\ T) : \Theta}$$

Fig. 13. A Selection of Type Rules for Statements

restricted to variables local to processes in P. Let the set of permanently halted processes in a global configuration with an environment ϵ and a channel μ be halted(ϵ, μ, P).

Lemma 1 states that the sequentialized program is an over-approximation of the parallel program with respect to halted processes. Intuitively, the lemma states that the sequentialized program can reach the same environment as the parallel program restricted to halted processes ($\rightarrow^*$ is the transitive closure over Global Semantics).

Lemma 1 (Rewrite Soundness). *Let P be a program in normal form. If*

- $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta', P'$
- *$P \circ P_0$ is symmetrically nondeterministic for some extension P_0*
- $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]$ *such that* $(\epsilon, \mu, P \circ P_0) \longrightarrow^* (\epsilon_F, \mu_F, F)$,

there exists $(\epsilon', \mu') \in [\![\Delta', \Gamma']\!]$ *such that* $(\epsilon', \mu', P' \circ P_0) \rightarrow^* (\epsilon_{F'}, \mu_{F'}, F')$ *and* $\epsilon_F|_H = \epsilon_{F'}|_H$ *where* $H = halted(\epsilon_F, \mu_F, F)$

The proof of Lemma 1 builds up on the proof of Theorem 4.1 in Bakst et al. [2017]. The main additions in our proof are the additional cases for the R-CondSend and R-CondReceive rewrite rules. Unlike the proof for R-Send and R-Receive, our proof must show that the sequentialized code can mirror the behavior of two different semantic rules depending on whether or not the message was successfully sent. The full proof is available in Appendix C.

5.2 Deadlock Freedom

The sequentialized program obtained using the rewrite rules in Figure 11 has a single process and does not contain any communication with other processes. As a result, it cannot deadlock. It follows from Lemma 1 that if a parallel program can be completely sequentialized, then it is also deadlock free. Using Lemma 1, we get the following proposition:

Proposition 1 (Transformations do not introduce deadlocks). *If P is the original program, P^A is the approximated program, $\varnothing, \varnothing, P \rightsquigarrow^* \varnothing, \Delta, \texttt{skip}$, and $\varnothing, \varnothing, P^A \rightsquigarrow^* \varnothing, \Delta', \texttt{skip}$, then the approximation itself does not introduce deadlocks in to the program.*

6 SAFETY ANALYSIS OF PARALLEL PROGRAMS

We show that our system has safety properties that ensure isolation of precise computations and type soundness. We use the sequentialization from Section 5 as an important building block for these analyses.

6.1 Approximate Type Analysis

Type System. We use similar type annotations as in EnerJ [Sampson et al. 2011]. We use type qualifiers to explicitly specify data that may be subject to approximations. We use a static type environment $\Theta : Var \mapsto T$ that maps variables to their type to check type-safety of statements.

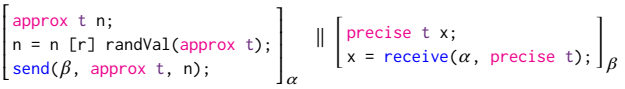


Fig. 14. An Example Program with Incorrect Type Annotations

We use two type judgments – (1) expressions are assigned a type, $\boxed{\Theta \vdash e : T}$, and (2) statements update the type environment, $\boxed{\Theta \vdash S : \Theta'}$. A selection of the typing rules are given in Figure 13.

6.2 Non-Interference

We show that the Parallely type system ensures non-interference between approximate data and precise data. Non-interference states that the results of precise data that need to be isolated from approximations will not be affected by changes in the approximate data. The *approx* type qualifiers are used to mark the data that can handle approximations and the type system guarantees non-interference.

The type system rules ensure that this property holds. Parallely only allows `cond-receive` statements to update approximate type variables (TR-CondReceive), therefore all `cond-send` statements can only communicate approximate type data (TR-CondSend). In addition, probabilistic choice statements can only update approximate type data (TR-Prob). Remaining rules are similar to those in EnerJ.

To prove this property for parallel programs, we start by defining non-interference for individual processes using the small-step semantic relation similar to EnerJ. As all communication channels in Parallely are typed, only precise data from a process can affect precise data in another process through communication. Therefore, by proving non-interference in each individual process, we can prove global non-interference.

We use $\cong$ to denote equivalence of frames, heaps, and channels limited to precise typed sections. Non-interference property states that starting at equivalent environments and executing a program will lead to equivalent final states regardless of differences in approximate data.

Theorem 1 (parallel non-interference). *Suppose* $\Theta \vdash P_i \parallel P_j : T$. $\forall \epsilon, \epsilon', \epsilon_f \in Env$ *and* $\mu, \mu', \mu_f \in Channel$, *s.t.,* $\langle \epsilon, \mu \rangle \cong \langle \epsilon', \mu' \rangle$, *if* $(\epsilon, \mu, P_i \parallel P_j) \longrightarrow_\psi (\epsilon_f, \mu_f, P_i' \parallel P_j)$, *then there exists* $\epsilon_f' \in Env$ *and* $\mu_f' \in Channel$ *such that* $(\epsilon', \mu', P_i \parallel P_j) \longrightarrow_\psi (\epsilon_f', \mu_f', P_i' \parallel P_j)$, *and* $\langle \epsilon_f, \mu_f \rangle \cong \langle \epsilon_f', \mu_f' \rangle$.

The proof of this property is analogous to the proofs of non interference in information flow security for parallel programs [Smith and Volpano 1998] and is provided in Appendix B.

While type checking individual processes allows us to prove non-interference between approximate and precise data, we need further checks to show that inter-process interactions don't cause deadlocks. Consider the two processes in Figure 14. While each process would pass our type checker (i.e., demonstrate non-interference), the program would deadlock as the two messaging channels do not match. The type of variable `x` in process β needs to be approximate for this program to function correctly. By incorporating canonical sequentialization to our safety analysis we can catch such bugs as the program would fail to sequentialize.

6.3 Type Soundness

Lemma 2 (The type system is sound for individual processes.). *For a single process, assuming there are no deadlocks, if* $\Theta \vdash s : \Theta'$, *then either* $\langle s, \sigma, h, \mu \rangle \rightharpoonup \langle \texttt{skip}, \sigma', h', \mu' \rangle$ *or* $\langle s, \sigma, h, \mu \rangle \rightharpoonup \langle s', \sigma', h', \mu' \rangle$ *and* $\Theta' \vdash s' : \Theta''$

Proof. (Sketch) The proof is by rule induction on typing rules (provided in Appendix B). □

Theorem 2 (The type system is sound.). *If* $\varnothing, \varnothing, P \rightsquigarrow^* \varnothing, \Delta, \texttt{skip}$ *and* $\Theta \vdash P : \Theta'$*, then either* $(\cdot,\cdot,P) \longrightarrow_\psi (\cdot,\cdot,\texttt{skip})$ *or* $(\cdot,\cdot,P) \longrightarrow_\psi (\cdot,\cdot,P')$ *and* $\Theta \vdash P' : \Theta'$

Proof. (Sketch) As the program P can be sequentialized, there are no deadlocks (Lemma 1). Therefore, there exists at least one individual process that is enabled and can take a step (i.e. the program makes progress). From Lemma 2 we know that this step will preserve the type of the statement and therefore the entire program will remain well typed. □

6.4 Relative Safety

Finally, we show how our approach can be used to prove relative safety of approximations. Relative safety allows us to transfer the reasoning about the safety of the original program to the approximated program [Carbin et al. 2012]. If the approximate transformation maintains relative safety, and the original program satisfies a property, then the transformed program also satisfies that property. For instance, if the original program has no array out of bound errors, and the approximation satisfies relative safety, then the approximate program is also has no array out of bound errors.

Definition 3 (Process-Local Relative safety [Carbin et al. 2013a]). *Let* P *be a program and* P^A *be the approximate program obtained by transforming* P*. The programs are relatively safe if when* $(\epsilon, \mu, P^A) \rightarrow^* (\epsilon_t, \mu_t, \mathit{assert}(e); \cdot)$ *there exists* ϵ_o *s.t.* $(\epsilon, \mu, P) \rightarrow^* (\epsilon_o, \mu_o, \mathit{assert}(e); \cdot)$ *and* $\forall x \in \mathit{free}(e) \cdot \epsilon_t(x) = \epsilon_o(x)$.

It states that if the approximate program satisfies (or does not satisfy) the property e, then there must exist an execution in the original program that satisfies (does not satisfy) the same property at the same program point. Therefore, if the assert statement is valid in the original program (i.e., its condition always evaluates to true), then it must also be valid in the transformed program. We can extend it to parallel computations: if any process-local safety properties are satisfied by the sequentialized program in its halting states, then they are also satisfied by the parallel program in its halting states, since according to Lemma 1 the sequentialized program is an over-approximation of the parallel program with respect to halted processes. We can immediately state the following proposition as a consequence of Lemma 1.

Proposition 2 (Relative safety of transformations via sequentialization). *If* P *is the original program,* P^A *is the program after applying some approximation,* $\varnothing, \varnothing, P \rightsquigarrow^* \varnothing, \Delta, \texttt{skip}$*,* $\varnothing, \varnothing, P^A \rightsquigarrow^* \varnothing, \Delta', \texttt{skip}$*, and a process-local safety property holds on the halting states of both* Δ *and* Δ'*, then the same safety property holds on the halting states of* P *and* P^A*.*

Therefore, we can use the sequentialized programs to prove the relative safety of approximations and that the original parallel programs will also satisfy relative safety.

7 RELIABILITY AND ACCURACY ANALYSIS OF PARALLEL PROGRAMS

In this section we define syntax for specifying reliability and accuracy requirements and show that we can generate guarantees on the parallel program by analyzing the canonical sequentializations.

7.1 Reliability Analysis – Semantic Foundations and Conditions

Reliability Predicates. Parallely can generate reliability predicates that characterize the reliability of a approximate program. A reliability predicate Q has the following form:

$$Q := R_f \leq R_f \mid Q \wedge Q$$
$$R_f := r \mid \mathcal{R}(O) \mid r \cdot \mathcal{R}(O)$$

A predicate can be a conjunction of predicates or a comparison between *reliability factors*. A reliability factor is either a rational number r, a *joint reliability factor*, or a product of a number and

a *joint reliability factor*. A reliability factor represents the probability that an approximate execution has the same values as the original execution for all variables in the set $O \subseteq \text{Var}$. By definition, $\mathcal{R}(\{\}) = 1$ (i.e., an empty set of variables has a reliability of 1).

For example, we can specify the constraint that the reliability of some variable x be higher than the constant 0.99 using the reliability predicate $0.99 \leq \mathcal{R}(\{x\})$. Intuitively, $\mathcal{R}(\{x\})$ refers to the probability that a approximate execution of the program has the same value for variable x as the exact execution of the program. A *joint reliability factor* such as $\mathcal{R}(\{x, y\})$ refers to the probability that both x and y have the same value.

We now define the semantics of reliability factors for our semantics by following the exposition in [Carbin et al. 2013b]. The denotation of a reliability factor $\boxed{[\![R_F]\!] \in \mathcal{P}(Env \times \Phi)}$ is the set of environment and environment distribution pairs that satisfy the predicate. An environment distribution $\phi \in \Phi = Env \mapsto \mathbb{R}$ is a probability distribution over possible approximate environments. For example, $[\![r]\!](\epsilon, \varphi) = r$. The denotation of $\mathcal{R}(O)$ is the probability that an environment ϵ_a sampled from Φ has the same value for all variables in O as the environment ϵ:

$$[\![\mathcal{R}(O)]\!](\epsilon, \varphi) = \sum_{\epsilon_u \in \mathcal{E}(O,\epsilon)} \varphi(\epsilon_u)$$

where, $\mathcal{E}(O, \epsilon)$ is the set of all environments in which the values of O are the same as in ϵ (which we express with the predicate *equiv*, formally defined in Carbin et al. [2013b, Section 5]):

$$\mathcal{E}(O, \epsilon) = \{\epsilon' \mid \epsilon' \in Env \wedge \forall v.v \in O \Rightarrow equiv(\epsilon, \epsilon', v)\}$$

Paired Execution Semantics. For reliability and accuracy analysis we define a *paired execution semantics* that couples an original execution of a program with an approximate execution, following the definition from Rely.

Definition 4 (Paired Execution Semantics [Carbin et al. 2013b]).

$$\langle\, s\,, \langle \epsilon, \varphi \rangle \rangle \Downarrow_\psi \langle \epsilon', \varphi' \rangle \text{ such that } \langle\, s\,, \epsilon \rangle \Longrightarrow_{1_\psi} \epsilon' \text{ and } \varphi'(\epsilon'_a) = \sum_{\epsilon_a \in \mathrm{E}} \varphi(\epsilon_a) \cdot p_a \text{ where } \langle\, s\,, \epsilon_a \rangle \overset{\cdot, p_a}{\Longrightarrow}_\psi \epsilon'_a$$

This relation states that from a configuration $\langle \epsilon, \varphi \rangle$ consisting of an environment ϵ and an *environment distribution* $\varphi \in \Phi$, the paired execution yields a new configuration $\langle \epsilon', \varphi' \rangle$. The execution reaches the environment ϵ' from the environment ϵ with probability 1 (expressed by the deterministic execution, 1_ψ). The environment distributions φ and φ' are probability mass functions that map an environment to the probability that the execution is in that environment. In particular, φ is a distribution on environments before the execution of s whereas φ' is the distribution on environments after executing s.

Reliability Transformer. Reliability predicates and the semantics of programs are connected through the view of a program as a reliability transformer.

Definition 5 (Reliability Transformer Relation [Carbin et al. 2013b]).

$$\boxed{\psi \models \{Q_{pre}\}\ s\ \{Q_{post}\}} \equiv \forall \epsilon, \varphi, \epsilon', \varphi'.\ (\epsilon, \varphi) \in [\![Q_{pre}]\!] \implies \langle s, \langle \epsilon, \varphi \rangle \rangle \Downarrow_\psi \langle \epsilon', \varphi' \rangle \implies (\epsilon', \varphi') \in [\![Q_{post}]\!]$$

Similar to the standard Hoare triple relation, if an environment and distribution pair $\langle \epsilon, \varphi \rangle$ satisfy a reliability predicate Q_{pre}, then the program's paired execution transforms them into a new pair $\langle \epsilon', \varphi' \rangle$ that satisfy a predicate Q_{post}.

Conditions for Analysis. For our reliability analysis to work, we require that the transformed approximate program P^A evaluated under exact execution semantics (1_ψ) is equivalent to the original program execution. We also need to ensure that the sequentialized program is equivalent (not just an over-approximation) in behavior to the parallel program. To achieve this property for our rewrite rules we removed sources of over-approximation from the language (e.g., Parallely does not support communication with external processes or wildcard receives from [Bakst et al. 2017]).

We perform the following transformations to simplify the analysis process: (1) we transform the program into its Single Static Assignment form; (2) we ensure that variable sets used in individual processes are disjoint by simple renaming; (3) we unroll finite loops; (4) we do conditional flattening as in Rely; and (5) we do not allow send and receive operations inside conditionals.

7.2 Reliability Analysis – Precondition Transformer

Given a reliability predicate that should hold after the execution of the program, the precondition generation produces a predicate that should hold before the execution of the program. The precondition generator starts at the end of the sequential program and successively builds preconditions, traversing the program backwards, and finally builds the precondition at the start of the program.

Substitution. We define substitution for reliability predicates the same way as Carbin et al. [2013b]. A substitution $e_0[e_2/e_1]$ replaces all occurrences of the expression e_1 with the expression e_2 within the expression e_0. The substitution matches set patterns. For instance, the pattern $R(\{x\} \cup X)$ represents a joint reliability factor that contains the variable x, alongside with the remaining variables in the set X. The substitution $r_1 \cdot \mathcal{R}(\{x, z\})[\mathcal{R}(\{y\} \cup X)/\mathcal{R}(\{x\} \cup X)]$ results in $r_1 \cdot \mathcal{R}(\{y, z\})$.

Precondition Generator. The reliability precondition generator is a function $\boxed{C \in S \times Q \mapsto Q}$ that takes as inputs a statement and a postcondition and produces a precondition as output. The analysis rules for instructions are the same as in Rely. We add the following rules to handle the new probabilistic choice and cast statements in Parallely:

$$
\begin{aligned}
C(\, x = e,\, Q\,) &= Q\,[\,\mathcal{R}(\,\rho(e) \cup X\,)\;/\;\mathcal{R}(\,\{x\} \cup X\,)\,] \\
C(\, x = e_1\,[r]\,e_2,\, Q\,) &= Q\,[\,r \cdot \mathcal{R}(\,\rho(e_1) \cup X\,)\;/\;\mathcal{R}(\,\{x\} \cup X\,)\,] \\
C(\, x = b?\, e_1 : e_2,\, Q\,) &= Q\,[\,\mathcal{R}(\,\rho(e_1) \cup \{b\} \cup X\,)\;/\;\mathcal{R}(\,\{x\} \cup X\,)\,] \\
C(\, x = (T)\, y,\, Q\,) &= Q\,[\,0\;/\;\mathcal{R}(\,\{x\} \cup X\,)\,]
\end{aligned}
$$

where $\rho(e)$ specifies the set of variables referred to by e. Intuitively, for simple assignment of an expression, any reliability specification containing x is updated such that x is replaced by the variables occurring in e. For probabilistic assignment, the reliability of x is equal to r times the reliability of variables occurring in e_1. For conditional assignment, where b is an integer variable, the reliability of x is equal to the reliability of variables occurring in e_1 and b; recall that e_1 is expected to be equivalent to the expression from the original program. Casting, which changes the precision of the variables causes the reliability of any variable to be 0, since in general reduced-precision values are not equal to the original values. The remaining rules are analogous to those from Carbin et al. [2013b, Section 5].

Figure 15 presents the results of the precondition generation for the sequential program given in Figure 12 (sequentialization of the example program from Section 3.3). Given the post condition $0.99 \leq \mathcal{R}(\{\beta.x\})$, which states that the reliability of β.x needs to be higher than 0.99, the precondition generation results in $0.99 \leq$ r. Solving these constraints is done via a subsumption-based decision procedure from Rely.

Our goal is to analyze the sequential version of the program and know that the generated reliability constraints are valid for the parallel program. In the following section we will discuss how to accomplish this goal through sequentialization.

```
{ 0.99 <= r }
α.n = 10;
{ 0.99 <= r ·R({α.n}) }
α.b = 1 [r] 0;
{ 0.99 <= R({α.n, α.b}) }
β.b = α.b ? 1 : 0;
{ 0.99 <= R({α.n, β.b}) }
β.x = β.b ? α.n : β.x
{ 0.99 <= R({β.x}) }
```

Fig. 15. An Example of the Reliability Precondition Generation.

7.3 Reliability Analysis via Canonical Sequentialization

In this section we will prove the following theorem stating that reliability transformer relations defined on a sequentialized approximate program (P^A_{seq}) will also hold on the original parallel approximate program (P^A).

Theorem 3 (Reliability Analysis Soundness). *If a program with approximation P^A (obtained by transforming the program P) can be sequentialized then the reliability analysis of the sequentialized program P^A_{seq} will also be valid for the parallel approximate program.*

$$\text{If } \varnothing, \varnothing, P^A \rightsquigarrow \varnothing, P^A_{seq}, \texttt{skip} \text{ then}$$
$$\psi \models \{Q_{pre}\}\ P^A_{seq}\ \{Q_{post}\} \implies \psi \models \{Q_{pre}\}\ P^A\ \{Q_{post}\}$$

We prove the theorem above in several steps. First, we show that any environment reached by the sequentialized Parallely program can be reached in the original parallel program (Lemma 3). Second, we use that result to prove that the original parallel program and the canonical sequentialization are equivalent (Lemma 4). Third, we show that approximate executions in the parallel program have equivalent executions in the sequential program (Lemma 5). Finally, we show that any paired execution of the parallel program has an equivalent execution in the sequentialized version (Lemma 6).

Our first step proves that any environment reached by the sequentialized program can be reached in the original parallel program. This is the converse of Lemma 1. Together these two lemmas allow us to establish that the two programs are equivalent.

Lemma 3. *Let P be a program in normal form. If*

- $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta', P'$
- $P \circ P_0$ *is symmetrically nondeterministic for some extension* P_0
- $(\epsilon', \mu') \in [\![\Delta', \Gamma']\!]$ *such that* $(\epsilon', \mu', P' \circ P_0) \rightarrow^* (\epsilon_{F'}, \mu_{F'}, F')$

there exists $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]$ *such that* $(\epsilon, \mu, P \circ P_0) \rightarrow^* (\epsilon_F, \mu_F, F)$ *and* $\epsilon_F|_H = \epsilon_{F'}|_H$ *where* $H = halted(\epsilon'_F, \mu'_F, F')$.

We prove Lemma 3 by induction on the derivation of Δ from P, splitting into multiple cases based on the rewriting rule. The full proof of Lemma 3 is available in the Appendix C.

Lemma 4 (equivalence of sequentialized program for halted states). *If* $\varnothing, \varnothing, P \rightsquigarrow^* \varnothing, \Delta, \texttt{skip}$ *then* $(\varnothing, \varnothing, P) \rightarrow^* (\epsilon_H, \varnothing, H)$ *if and only if* $(\varnothing, \varnothing, \Delta) \rightarrow^* (\epsilon'_H, \varnothing, H')$ *such that* $\epsilon_H = \epsilon'_H$ *where all processes are permanently halted in* $(\epsilon_H, \varnothing, H)$.

The proof of Lemma 4 is by using Lemma 1 and Lemma 3, which state that the sequentialized program is an over-approximation of the parallel program and that the parallel program is an over-approximation of the sequential program respectively (with respect to halted processes). Thus, the sequentialized program is equivalent to the parallel program with respect to halted processes.

In the following lemma we show that approximations have the same behaviors on the parallel and the sequentialized programs:

Lemma 5 (Aggregate Semantics equivalence).

$$\text{If } \varnothing, \varnothing, P \rightsquigarrow \varnothing, \Delta, \texttt{skip} \text{ then}$$
$$\langle P,\ \epsilon\rangle \overset{p}{\Longrightarrow}_\psi \epsilon' \text{ if and only if } \langle \Delta,\ \epsilon\rangle \overset{p}{\Longrightarrow}_\psi \epsilon'$$

Proof. (Sketch) From rewrite soundness lemma (Lemma 4), we know that $(S, \varnothing, \epsilon) \rightarrow^*_\psi (\texttt{skip}, \varnothing, \epsilon')$ if and only if $(\Delta, \varnothing, \epsilon) \rightarrow^*_\psi (\texttt{skip}, \varnothing, \epsilon')$. Probabilistic differences in executions only appear in Parallely programs in the form of probabilistic choice statements and cast statements. All such statements

are sequentialized without any change and added straight to the sequential prefix through the `R-Context` rule.

In addition, all of the probabilistic transitions we model are independent of the execution environment. Therefore, aggregated over all possible schedules, the probability of reaching the same final environment will be the same for the two versions of the program. □

Finally, the next lemma states that sequentialization preserves the paired execution relation.

Lemma 6 (Rewrites preserve paired executions).

$$\textit{If } \varnothing, \varnothing, P \rightsquigarrow \varnothing, \Delta, \texttt{skip} \textit{ then}$$
$$\langle\ \Delta\ , \langle \epsilon, \varphi \rangle\rangle \Downarrow_\psi \langle \epsilon', \varphi' \rangle \iff \langle\ P\ , \langle \epsilon, \varphi \rangle\rangle \Downarrow_\psi \langle \epsilon', \varphi' \rangle$$

Proof. From Lemma 4, $\langle \epsilon,\ \varnothing,\ \ \Delta\ \rangle \overset{1}{\Longrightarrow}_{1_\psi} \epsilon'$ iff $\langle \epsilon_a,\ \varnothing,\ \ P\ \rangle \overset{1}{\Longrightarrow}_{1_\psi} \epsilon'$. In addition, from Lemma 5, $\langle \epsilon_a,\ \varnothing,\ \ \Delta\ \rangle \overset{p_a}{\Longrightarrow}_\psi \epsilon'_a$ iff $\langle \epsilon_a,\ \varnothing,\ \ P\ \rangle \overset{p_a}{\Longrightarrow}_\psi \epsilon'_a$. Therefore, $\sum_{\epsilon_u \in \mathcal{E}(O,\epsilon)} \varphi(\epsilon_u) \cdot p_a$ is the same for both versions of the program, leading to the same distributions. □

We can now use these lemmas to prove our main theorem:

Proof of Theorem 3. Since $\psi \models \{Q_{pre}\}\ P^A_{seq}\ \{Q_{post}\}$, we know that,

$$\forall \langle \epsilon, \varphi \rangle \in [\![Q_{pre}]\!],\ \langle P^A_{seq}, \langle \epsilon, \varphi \rangle\rangle \Downarrow_\psi \langle \epsilon', \varphi' \rangle \implies (\epsilon', \varphi') \in [\![Q_{post}]\!].$$

Lemma 6 states that both P^A and P^A_{seq} have the same paired execution behavior. Consequently, $\langle\ P^A_{seq}\ , \langle \epsilon, \varphi \rangle\rangle \Downarrow_\psi \langle \epsilon', \varphi' \rangle$ if and only if $\langle\ P^A\ , \langle \epsilon, \varphi \rangle\rangle \Downarrow_\psi \langle \epsilon', \varphi' \rangle$. It follows that if $\langle P^A_{seq}, \langle \epsilon, \varphi \rangle\rangle \Downarrow_\psi \langle \epsilon', \varphi' \rangle \implies (\epsilon', \varphi') \in [\![Q_{post}]\!]$, then $\langle P^A, \langle \epsilon, \varphi \rangle\rangle \Downarrow_\psi \langle \epsilon', \varphi' \rangle$ and $(\epsilon', \varphi') \in [\![Q_{post}]\!]$. Therefore, we conclude, $\forall \langle \epsilon, \varphi \rangle \in [\![Q_{pre}]\!],\ \langle P^A, \langle \epsilon, \varphi \rangle\rangle \Downarrow_\psi \langle \epsilon', \varphi' \rangle \implies (\epsilon', \varphi') \in [\![Q_{post}]\!]$.

□

7.4 Accuracy Analysis

Parallely can generate an accuracy predicate that characterizes the magnitude of error of an approximate program. An accuracy predicate A has the following form ([Misailovic et al. 2014]):

$$A := D \leq D \mid A \wedge A$$
$$D := r \mid \mathcal{D}(x) \mid r \cdot \mathcal{D}(x)$$

An accuracy predicate can be a conjunction of predicates or a comparison between two error factors. An error factor can be a rational number r or the maximum error associated with a program variable x. The user can use an accuracy predicate to specify the maximum allowed error of various program variables at the end of execution. Parallely's analysis generates a precondition accuracy predicate that must hold at the start of the program for the user specified accuracy predicate to hold at the end of the program. To assist with error calculation, the user must specify the intervals of values that each input program variable can take. For a program variable x, this interval is specified as $x[a, b]$, indicating that the variable x can take any value in the range $[a, b]$ at the start of the program. Parallely then performs interval analysis similar to Chisel [Misailovic et al. 2014] to generate the accuracy precondition.

Theorem 4. *If the program with approximation P^A (obtained by transforming the program P) can be canonically sequentialized, then the accuracy analysis of the sequentialized program P^A_{seq} will also be valid for the parallel approximate program.*

$$\textit{If } \varnothing, \varnothing, P^A \rightsquigarrow \varnothing, P^A_{seq}, \texttt{skip} \textit{ then}$$
$$\psi \models \{A_{pre}\}\ P^A_{seq}\ \{A_{post}\} \implies \psi \models \{A_{pre}\}\ P^A\ \{A_{post}\}$$

The proof follows directly from Lemma 6 and is analogous to the proof of Theorem 3.

Table 1. Accuracy and Performance of Approximated Programs

Benchmark	Parallel Pattern	Approximation	LoC	LoC Changed	Types Changed	Accuracy Property
PageRank	Map	Failing Tasks	49	12	4	$0.99 \leq \mathcal{R}(\text{result})$
Scale	Map	Failing Tasks	82	10	4	$0.99 \leq \mathcal{R}(\text{result})$
Blackscholes	Map	Noisy Channel	115	6	3	$0.99 \leq \mathcal{R}(\text{result})$
SSSP	Scatter-Gather	Noisy Channel	70	14	6	$0.99 \leq \mathcal{R}(\text{result})$
BFS	Scatter-Gather	Noisy Channel	70	14	6	$0.99 \leq \mathcal{R}(\text{result})$
SOR	Stencil	Precision Reduction	50	16	6	$10^{-6} \geq \mathcal{D}(\text{result})$
Motion	Map/Reduce	Approximate Reduce	43	13	6	-
Sobel	Stencil	Precision Reduction	46	16	6	$10^{-6} \geq \mathcal{D}(\text{result})$

8 EVALUATION

To evaluate Parallely expressiveness and efficiency, we implemented a set of benchmarks from several application domains. These benchmarks exhibit diverse parallel patterns, can tolerate error in the output and have been studied in the approximate computing literature:

- *PageRank:* Computes PageRank for nodes in a graph [Page et al. 1999]. The parallel version is derived from the CRONO benchmark suite [Ahmad et al. 2015]. We verify the reliability of the calculated PageRank array.
- *Scale:* Computes a bigger version of an image. The version is derived from a Chisel benchmark [Misailovic et al. 2014]. We verify the reliability of the output image.
- *Blackscholes:* Computes the prices of a portfolio of options. The version is derived from PARSEC suite [Bienia 2011]. We verify the reliability of the array of calculated option prices.
- *SSSP:* Single Source Shortest Path in a graph. The version is derived from CRONO benchmark suite [Ahmad et al. 2015]. We verify the reliability of the calculated distances array.
- *BFS:* Breadth first search in a graph. The version is also derived from the CRONO suite [Ahmad et al. 2015]. We verify the reliability of the array of visited vertices.
- *SOR:* A kernel for computing successive over-relaxation (SOR). The version is derived from Chisel benchmark [Misailovic et al. 2014]. We verify the accuracy of the resultant 2D array.
- *Motion:* A pixel-block search algorithm from the x264 video encoder. The version is derived from Rely benchmark [Carbin et al. 2013b]. Although the accuracy specification is out of Parallely's reach, we still verify type safety, non-interference, and deadlock-freeness.
- *Sobel:* Sobel edge-detection filter calculation. We use the version from AxBench [Yazdanbakhsh et al. 2017]. We verify the accuracy of the output image.

Table 1 presents the summary of the benchmarks and their specifications. For each benchmark, it presents the parallel pattern and approximation (Section 3), the lines of Parallely code, the number of lines and type declarations affected by approximation, and the accuracy property we check. We omit the accuracy specification for Motion since it returns an index. For the accuracy analysis we assumed that the inputs to SOR and Sobel are in the range $[0, 1]$.

For each kernel from Section 3 and benchmark listed above, we measured the real time required to perform sequentialization, type checking, and reliability/accuracy checking. We ran the experiments on an Intel Xeon E5-1650 v4 processor with 32 GB of RAM. We implemented our parser using ANTLR and the other modules in Python.

Table 2. Performance of Parallely Analysis for Kernels (left) and Benchmarks (right)

Kernels	T_{Seq}	T_{Safety}	$T_{Rel/Acc}$
Simple (precision)	0.4 ms	0.2 ms	15 ms
Simple (noisy)	0.3 ms	0.2 ms	14 ms
Simple (failing tasks)	0.4 ms	0.2 ms	17 ms
Map (memoization)	0.9 ms	0.4 ms	48 ms
Reduce (sampling)	0.9 ms	0.5 ms	53 ms
Scan (noisy)	1.9 ms	0.9 ms	76 ms
Partition (failing tasks)	1.2 ms	0.6 ms	17 ms
Stencil (precision)	1.4 ms	0.7 ms	73 ms
Average	0.9 ms	0.5 ms	39 ms

Benchmarks	T_{Seq}	T_{Safety}	$T_{Rel/Acc}$
PageRank	1.8 s	1 ms	168 s
Scale	6.5 s	3 ms	7.4 s
Blackscholes	0.2 s	1 ms	12 s
SSSP	9.6 s	6 ms	9.6 s
BFS	8.9 s	6 ms	9.2 s
SOR	8.3 s	4 ms	53 s
Motion	2.9 s	1 ms	–
Sobel	0.2 s	6 ms	72 s
Average	4.8 s	3 ms	47.3 s

8.1 Efficiency of Parallely

Table 2 presents a summary of Parallely's analysis for the real world programs and kernels from Section 3. For each benchmark, Column 2 presents the time for sequentialization, Column 3 presents the time for type safety analysis, and Column 4 presents the time for reliability analysis.

Overall, Parallely was efficient for both kernels and benchmark applications. The sequentialization analysis took only a few seconds, even for more complex benchmarks. Type checking was instantaneous – showcasing the benefits of our design to first do per-process type checking and then sequentialization (which is more expensive). Therefore, if there are simple type errors, they can be easily discovered and reported to the developer. The time for the reliability analysis is proportional to the amount of computation in each benchmark. The PageRank benchmark is an outlier, with 168 seconds. Most of this time was spent on analyzing unrolled loops in the sequentialized version of the program.

8.2 Benefits of Approximations

To analyze the benefits of the approximate transformations we translated the programs in Parallely to the Go language and measured the speedup. We only looked at the effect on runtime for three approximations: failing tasks, precision reduction, and approximate reduce. We selected representative inputs and present the average results over 100 runs (for statistical significance). Noisy channel evaluation requires detailed hardware simulation, which is out of our scope.

For each benchmark, Table 3 shows the approximations applied (Column 2), the metric used to compare the accuracy of the final results (Column 3), the speedup obtained using approximations, and the errors calculated using the relevant metric (Column 4). We next describe how we simulated each approximation. More details about the evaluation are available in Appendix E.

- **Failing Tasks**. We simulate this approximation by letting child processes fail with a small random chance. If a child process fails, it sends a null signal to the main process. Many calculations can tolerate a small number of incorrect calculations without a significant loss of accuracy, such as the PageRank and Scale benchmarks. In the precise version, if the main process observes that a child process has failed, the child process is restarted.
- **Precision Reduction**. We reduce the precision of data sent to the worker processes from 64 bit floats to 32 bit floats. Likewise, the workers send the results as 32 bit floats. The results are converted back into 64 bit floats in the main process. Reducing precision only affects the least significant bits of the variables, so the accuracy degradation is acceptable for many calculations. In the precise version, this reduction in precision is not performed.
- **Approximate Reduce**. We simulate this approximation by letting worker processes decide not to do any work and return a null signal to the main process. The main process aggregates the

Table 3. Performance Benefits from Approximations

Benchmark	Approximation	Accuracy metric	Speedup	End-to-End Error
PageRank	Failing Tasks	Avg. difference in PageRank	1.09x	2.87×10^{-8}
Scale	Failing Tasks	PSNR	1.35x	38.80 dB
SOR	Precision Reduction	Avg. sum of squared diffs. (SSD)	1.76x	3.9×10^{-16}
Motion	Approximate Reduce	Avg. difference from best SSD	1.20x	0.09
Sobel	Precision Reduction	Avg. sum of squared diffs. (SSD)	1.70x	1.56×10^{-16}

received (non-null) results and adjusts the aggregate according to the number of returned results. In our evaluation each worker process only does work 10% of the time. When a large number of similar calculation results are aggregated by a reduction operation (such as sum, minimum, or maximum), the result of performing the calculations on a subset of data is often very close to the result of performing the calculation on the entire dataset. In the exact version, worker processes always return a result.

Results. Table 3 shows the speedup and error obtained as a result of the approximations. Column 1 shows the benchmark, Column 2 shows the approximation applied, Column 3 shows the error metric, Column 4 shows the speedup with respect to the precise version, and Column 5 shows the average measured error. For benchmarks with failing tasks (PageRank and Scale), the average is taken over runs that experienced at least one task failure. For Motion, SOR, and Sobel, the average is taken over all runs.

The results show that Parallely can be used to generate approximate versions of programs with significant performance improvements, while providing important safety guarantees. For PageRank and Scale, Parallely verified a reliability specification. However, even when a task fails, the error in the final result is very low. This result shows that programs with high reliability also often have high accuracy. For Motion, even when skipping most of tasks, the calculated minimum SSD is very close to the actual minimum SSD. For SOR and Sobel, the actual error is significantly lower than the worst case bound of 10^{-6} verified by Parallely.

9 RELATED WORK

Approximate Program Analyses. While approximations have been justified predominantly empirically (e.g., [Rinard 2006, 2007; Baek and Chilimbi 2010; Misailovic et al. 2010; Ansel et al. 2011; Hoffmann et al. 2011; Sidiroglou et al. 2011; Misailovic et al. 2013; Rubio-González et al. 2013; Ansel et al. 2014; Samadi et al. 2014; Schkufza et al. 2014; Achour and Rinard 2015; Ding et al. 2015; Mitra et al. 2017; Nongpoh et al. 2017; Xu et al. 2018; Fernando et al. 2019c; Joshi et al. 2019]), several sound static analyses emerged in recent years. EnerJ [Sampson et al. 2011, 2015] presents an information-flow type system that separates approximate and precise data. As this *non-interference* constraint is restrictive, EnerJ allows approximate data to influence precise data through unsound type conversion (*endorse*). While we did not use such conversions, a general approach that more rigorously reasons about type conversions is an interesting topic for future work.

Carbin et al. [2012], Carbin et al. [2013a], He et al. [2018], and Boston et al. [2018] present frameworks for reasoning about relational safety properties and relative safety. For accuracy and reliability, researchers have proposed quantitative analyses [Gaffar et al. 2002; Osborne et al. 2007; Chaudhuri et al. 2011; Misailovic et al. 2011; Zhu et al. 2012; Carbin et al. 2013a; Misailovic et al. 2014; Canino and Liu 2017; Darulova et al. 2018; Lidman and Mckee 2018]. Some (e.g., [Chaudhuri et al. 2011; Misailovic et al. 2011; Zhu et al. 2012]) focus on specific sequential transformations and code patterns. For general programs, Rely [Carbin et al. 2013a], Chisel [Misailovic et al. 2014], and Decaf [Boston et al. 2015] analyze quantitative reliability and/or accuracy when running on

unreliable hardware. Approaches such as Darulova et al. [2018], Chiang et al. [2017] and [Magron et al. 2017] reason about floating point errors in programs.

All these approaches have been developed for sequential programs. Parallely generalizes several key analyses for message-passing programs and presents a methodology for proving the correctness of the analysis via canonical sequentialization. Parallely's language further presented general approximation constructs and quantitative reasoning about both software and hardware approximations, in contrast to the previous approaches that were closely coupled with the hardware-specific error models [Carbin et al. 2013a; Misailovic et al. 2014; Boston et al. 2015].

Analysis of Parallel Programs. Previous work discusses the verification of programs using message passing interfaces, e.g., [Siegel 2005; Siegel and Avrunin 2005; Siegel and Gopalakrishnan 2011; Michael et al. 2019]. Various tools are available to do model checking for Erlang, a popular actor language [Huch 1999; Fredlund and Svensson 2007; D'Osualdo et al. 2013]. However, actor languages often have only one incoming message queue, making it difficult to prove properties about them via canonical sequentialization. Alternatively, one can reduce complex parallel programs into relatively simpler programs, or use representative program traces that are sufficient for reasoning about the properties of the original parallel program [Lipton 1975; Godefroid 1996; Flanagan and Godefroid 2005; Abdulla et al. 2014; Desai et al. 2014]. Sequentialization approaches such as Lal and Reps [2008]; La Torre et al. [2009] reduce parallel programs to sequential versions to provide bounded guarantees. Other approaches [Blom et al. 2015; Huisman 2017] allow developers to annotate a sequential program and verify that automated parallelizations are equivalent.

Our work primarily draws inspiration from Canonical Sequentialization by Bakst et. al. [Bakst et al. 2017], which enables verification of general safety properties of parallel functional programs. We show that canonical sequentialization is a solid foundation for reasoning about approximate programs. We anticipate that future progress on sequentialization such as Gleissenthall et al. [2019], can provide new opportunities for precisely modeling and analyzing various approximations.

Foundations of Parallel Programming. Researchers have defined various formalisms for representing parallel computation, e.g., Actor model [Agha and Hewitt 1985; Agha 1986], Pi calculus [Milner 1999], Petri nets [Peterson 1977] and others. To make an approximation-aware analyses both tractable and easier to express/implement, we aimed for a modular approach that separates reasoning about concurrency from reasoning about quantitative properties. Canonical sequentialization proved to be a good match in this regard, while offering a good level of generality and reusing the formalizations of the analyses for sequential programs. An alternative route would be to define individual analyses, such as EnerJ, Rely, or Chisel directly on a parallel calculus. While such an approach would be equally fruitful for the property in question, it would require reasoning about interleavings and shared data in an ad-hoc manner and new proof would need to be derived for every other analysis, making it hard to support multiple advanced analyses.

10 CONCLUSION

We presented Parallely, a language and system for verification of approximations in parallel message-passing programs. It is the first approach tailored to systematically represent and analyze approximate parallel computations. In this paper, we have presented how to leverage a large body of techniques for verification of approximate sequential programs to the parallel setting, while allowing us to generalize those and increase their reach (as in the case of Rely). Our experimental results on a set of approximate computational kernels, representative transformations, and full programs show that Parallely's analysis is both fully automatable and efficient.

Our approximation-aware canonical sequentialization is particularly promising: in addition to the accuracy analyses that we studied in this paper, we anticipate that other existing and future

accuracy analyses of sequential programs can directly leverage sequentialization to analyze parallel computations. We anticipate that our theoretical results will also be useful to reason about other quantitative properties of parallel programs, such as differential privacy or fairness.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for useful suggestions. The research presented in this paper was supported in part by NSF Grants CCF-1629431, CCF-1703637, and CCF-1846354, and DARPA Domain-specific Systems on Chip (DSSOC) program, part of the Electronics Resurgence Initiative (ERI), under Contract No. HR0011-18-C-0122.

REFERENCES

R. J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18 (1975).
J. L. Peterson. 1977. Petri nets. *ACM Computing Surveys (CSUR)* 3 (1977).
G. Agha and C. Hewitt. 1985. *Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism.* Technical Report. Cambridge, MA, USA.
G. Agha. 1986. An Overview of Actor Languages. In *OOPWORK*.
P. Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag.
G. Smith and D. Volpano. 1998. Secure Information Flow in a Multi-threaded Imperative Language. In *POPL*.
F. Huch. 1999. *Verification of Erlang programs using abstract interpretation and model checking.*
R. Milner. 1999. *Communicating and mobile systems: the pi calculus.* Cambridge university press.
L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report.
A. Gaffar, O. Mencer, W. Luk, P. Cheung, and N. Shirazi. 2002. Floating-point bitwidth analysis via automatic differentiation. In *FPT*.
J. Dean and S. Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. *OSDI* (2004).
C. Flanagan and P. Godefroid. 2005. Dynamic Partial-order Reduction for Model Checking Software. In *POPL*.
S. F. Siegel. 2005. Efficient Verification of Halting Properties for MPI Programs with Wildcard Receives. In *VMCAI*.
S. F. Siegel and G. S. Avrunin. 2005. Modeling Wildcard-free MPI Programs for Verification. In *PPoPP*.
M. Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*.
L.-Å. Fredlund and H. Svensson. 2007. McErlang: a model checker for a distributed functional programming language. In *ICFP*.
W. Osborne, R. Cheung, J. Coutinho, W. Luk, and O. Mencer. 2007. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. In *FPL*.
M. Rinard. 2007. Using Early Phase Termination to Eliminate Load Imbalances at Barrier Synchronization Points. In *OOPSLA*.
A. Lal and T. Reps. 2008. Reducing concurrent analysis under a context bound to sequential analysis. In *International Conference on Computer Aided Verification*. 37–51.
S. Chakradhar, A. Raghunathan, and J. Meng. 2009. Best-Effort Parallel Execution Framework for Recognition and Mining Applications. In *IPDPS*.
S. La Torre, P. Madhusudan, and G. Parlato. 2009. Reducing context-bounded concurrent reachability to sequential reachability. In *International Conference on Computer Aided Verification*. 477–492.
W. Baek and T. M. Chilimbi. 2010. Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation. In *PLDI*.
S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. 2010. Quality of service profiling. In *ICSE*.
J. Ansel, Y. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *CGO*.
C. Bienia. 2011. *Benchmarking modern multiprocessors.*
S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. 2011. Proving Programs Robust. In *ESEC/FSE*.
H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. 2011. Dynamic Knobs for Responsive Power-Aware Computing. In *ASPLOS*.
S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. 2011. Flikker: saving DRAM refresh-power through critical data partitioning. (2011).
S. Misailovic, D. Roy, and M. Rinard. 2011. Probabilistically Accurate Program Transformations. In *SAS*.
B. Recht, C. Re, S. Wright, and F. Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*.

A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*.
S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. 2011. Managing Performance vs. Accuracy Trade-offs With Loop Perforation. In *FSE*.
S. F. Siegel and G. Gopalakrishnan. 2011. Formal Analysis of Message Passing. In *VMCAI*.
A. Udupa, K. Rajan, and W. Thies. 2011. ALTER: Exploiting Breakable Dependences for Parallelization. In *PLDI*.
M. Carbin, D. Kim, S. Misailovic, and M. Rinard. 2012. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. In *PLDI*.
L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener. 2012. Programming with relaxed synchronization. In *Relax Workshop*.
Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. 2012. Randomized Accuracy-Aware Program Transformations for Efficient Approximate Computations. In *POPL*.
M. Carbin, D. Kim, S. Misailovic, and M. Rinard. 2013a. Verified integrity properties for safe approximate program transformations. In *PEPM*.
M. Carbin, S. Misailovic, and M. C. Rinard. 2013b. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *OOPSLA*.
E. D'Osualdo, J. Kochems, and C.-H. L. Ong. 2013. Automatic verification of Erlang-style concurrency. In *International Static Analysis Symposium*.
S. Misailovic, D. Kim, and M. Rinard. 2013. Parallelizing Sequential Programs With Statistical Accuracy Tests. *ACM TECS Special Issue on Probabilistic Embedded Computing* (2013).
C. Rubio-González, C. Nguyen, H. Nguyen, J. Demmel, W. Kahan, K. Sen, D. Bailey, C. Iancu, and D. Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *SC*.
P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. 2014. Optimal Dynamic Partial Order Reduction. In *POPL*.
J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. M. O'Reilly, and S. Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *PACT*.
A. Desai, P. Garg, and P. Madhusudan. 2014. Natural Proofs for Asynchronous Programs Using Almost-synchronous Reductions. In *OOPSLA*.
S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. 2014. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *OOPSLA*.
M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. 2014. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *ASPLOS*.
E. Schkufza, R. Sharma, and A. Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*.
S. Achour and M. Rinard. 2015. Energy Efficient Approximate Computation with Topaz. In *OOPSLA*.
M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. 2015. CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In *IISWC*.
S. Blom, S. Darabi, and M. Huisman. 2015. Verification of loop parallelisations. In *International Conference on Fundamental Approaches to Software Engineering*. 202–217.
B. Boston, A. Sampson, D. Grossman, and L. Ceze. 2015. Probability type inference for flexible approximate programming. (2015).
S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks. 2015. HELIX-UP: Relaxing program semantics to unleash parallelization. In *CGO*.
Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U. M. O'Reilly, and S. Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. In *PLDI*.
I. Goiri, R. Bianchini, S. Nagarakatte, and T. Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *ASPLOS*.
A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin. 2015. *ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing*. Technical Report.
M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*.
R. Akram, M. M. U. Alam, and A. Muzahid. 2016. Approximate Lock: Trading off Accuracy for Performance by Skipping Critical Sections. In *ISSRE*.
A. Bakst, K. v. Gleissenthall, R. G. Kici, and R. Jhala. 2017. Verifying Distributed Programs via Canonical Sequentialization. In *OOPSLA*.
R. Boyapati, J. Huang, P. Majumder, K. H. Yum, and E. J. Kim. 2017. APPROX-NoC: A Data Approximation Framework for Network-On-Chip Architectures. In *ISCA*.
A. Canino and Y. D. Liu. 2017. Proactive and Adaptive Energy-aware Programming with Mixed Typechecking. In *PLDI*.

W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić. 2017. Rigorous floating-point mixed-precision tuning. In *POPL*.

M. Huisman. 2017. A Verification Technique for Deterministic Parallel Programs. In *PPDP*.

V. Magron, G. Constantinides, and A. Donaldson. 2017. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Trans. Math. Software* 43, 4 (Jan. 2017).

S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi. 2017. Phase-aware optimization in approximate computing. In *CGO*.

B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee. 2017. AutoSense: A Framework for Automated Sensitivity Analysis of Program Data. *IEEE Transactions on Software Engineering* 43 (2017).

A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design Test* 34, 2 (April 2017).

F. Betzel, K. Khatamifard, H. Suresh, D. J. Lilja, J. Sartori, and U. Karpuzcu. 2018. Approximate Communication: Techniques for Reducing Communication Bottlenecks in Large-Scale Parallel Systems. *ACM Computing Surveys (CSUR)* 51 (2018).

B. Boston, Z. Gong, and M. Carbin. 2018. Leto: verifying application-specific hardware fault tolerance with programmable execution models. In *OOPSLA*.

E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian. 2018. Daisy-Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *TACAS*.

E. A. Deiana, V. St-Amour, P. A. Dinda, N. Hardavellas, and S. Campanoni. 2018. Unconventional Parallelization of Nondeterministic Applications. In *ASPLOS*.

S. He, S. K. Lahiri, and Z. Rakamarić. 2018. Verifying relative safety, accuracy, and termination for program approximations. *Journal of Automated Reasoning* 60, 1 (2018).

S. K. Khatamifard, I. Akturk, and U. R. Karpuzcu. 2018. On Approximate Speculative Lock Elision. *IEEE Transactions on Multi-Scale Computing Systems* 2 (2018).

J. Lidman and S. A. Mckee. 2018. Verifying Reliability Properties Using the Hyperball Abstract Domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 40, 1 (2018), 3.

P. Stanley-Marbell and M. Rinard. 2018. Perceived-Color Approximation Transforms for Programs that Draw. *IEEE Micro* 38, 4 (2018), 20–29.

J. R. Stevens, A. Ranjan, and A. Raghunathan. 2018. AxBA: an approximate bus architecture framework. In *ICCAD*.

R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, S. Misailovic, and S. Bagchi. 2018. Videochef: efficient approximation for streaming video processing pipelines. In *USENIX ATC*.

V. Fernando, A. Franques, S. Abadal, S. Misailovic, and J. Torrellas. 2019a. Replica: A Wireless Manycore for Communication-Intensive and Approximate Data. In *ASPLOS*.

V. Fernando, K. Joshi, D. Marinov, and S. Misailovic. 2019c. Identifying Optimal Parameters for Randomized Approximate Algorithms. In *Workshop on Approximate Computing Across the Stack*.

V. Fernando, K. Joshi, and S. Misailovic. 2019b. Appendix to Parallely `https://vimuth.github.io/parallely/appendix.pdf`.

K. Gleissenthall, R. G. Kici, A. Bakst, D. Stefan, and R. Jhala. 2019. Pretend Synchrony. In *POPL*.

K. Joshi, V. Fernando, and S. Misailovic. 2019. Statistical Algorithmic Profiling for Randomized Approximate Programs. In *ICSE*.

E. Michael, D. Woos, T. Anderson, M. D. Ernst, and Z. Tatlock. 2019. Teaching Rigorous Distributed Systems With Efficient Model Checking. In *EuroSys*.