# Towards Verified Scalable Parallel Computing with Coq & Spark

Frédéric Loulergue
Univ. Orléans, INSA CVL, LIFO, EA 4022
Orléans, France
frederic.loulergue@univ-orleans.fr

Jolan Philippe
IMT Atlantique, Inria, LS2N (UMR CNRS 6004)
Nantes, France
jolan.philippe@imt-atlantique.fr

## ABSTRACT

SyDPaCC (Systematic Development of programs for Parallel and Cloud Computing) is a framework for the Coq interactive theorem prover. It allows to systematically develop correct parallel programs from specifications via verified and automated program transformations. The obtained programs are scalable, i.e. able to run on numerous processors. SyDPaCC produces programs written in the multi-paradigm and functional programming language OCaml with calls to the BSML (Bulk Synchronous parallel ML) parallel programming library. In this paper we present ongoing work towards an extension of SyDPaCC to be able to produce Scala programs using Apache Spark for parallel processing.

## CCS CONCEPTS

• **Computing methodologies → MapReduce algorithms**; • **Software and its engineering → Software verification**.

## KEYWORDS

Interactive theorem proving, Program transformation, Coq, Scala, Apache Spark, Algorithmic skeletons, Functional programming

**ACM Reference Format:**
Frédéric Loulergue and Jolan Philippe. 2023. Towards Verified Scalable Parallel Computing with Coq & Spark. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '23), July 18, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3605156.3606450

## 1 INTRODUCTION

For complex software such as operating systems or compilers, there exist verified implementations [22, 25, 43] using interactive theorem provers, in these examples Isabelle/HOL [35], Coq [4, 40, 45] and HOL Light respectively. Although it may not be trivial to understand very precisely what are the guarantees provided by such formal developments [34], they indeed offer code without bugs [47].

In the domain of scalable parallel computing (from a dozen of cores to potentially a dozen of thousands or more), SyDPaCC [31, 32] is a framework for Coq that supports the systematic development of correct parallel programs from specifications via verified and automated program transformations. Currently, SyDPaCC provides transformations based on list homomorphism theorems [14] and the diffusion theorem [20] which can be seen as an extension of the first homomorphism theorem to functions taking an accumulating parameter in addition to the list parameter. SyDPaCC also offers proofs of correspondences between sequential higher-order functions and algorithmic skeletons [7] that can be considered as higher-order functions implemented in parallel. These skeletons are implemented using the parallel primitives of BSML [29, 44] a library for scalable parallel programming with the multi-paradigm and functional programming language OCaml [26]. From its design, SyDPaCC was supposed to offer several backends, not only BSML+OCaml. The contribution of this paper is ongoing work to extend SyDPaCC to be able to produce Scala [36] programs using Apache Spark [2] for parallel processing.

The rest of the paper is organized as follows. Section 2 gives an overview of the features of Coq, BSML and Spark. Section 3 presents the principles of the SyDPaCC approach. Section 4 is devoted to the extension of SyDPaCC to deal with Scala+Spark programs in the context of list homomorphisms. Section 5 discusses a path towards supporting transformations based on the diffusion theorem. We compare our proposal to the state-of-the-art in Section 6. We conclude and identify future research directions in Section 7.

## 2 BACKGROUND

### 2.1 The Coq Proof Assistant

Coq [4, 40, 45] is an interactive theorem prover. It can be seen as a functional programming language, similar in syntax to languages such as Standard ML, F# or OCaml. It is based on the Curry-Howard correspondance [18] where types correspond to mathematical statements and programs correspond to proofs. All functions should be terminating and total, otherwise non-terminating or incomplete proofs could exist. The type system of Coq is much more expresssive than the type systems of the programming languages cited above. While directly writing proofs as functional programs is possible, in practice a language of so-called tactics is used to build proof terms.

Figure 1 presents in lines 1–7 the definition of a tail recursive version of `map`, named `map'`, that uses the auxiliary recursive function `map_aux` and the pre-defined `List.rev'` function that reverses a list (`rev'` is a tail recursive function). Both functions are polymorphic with type arguments `A` and `B` which are implicit in the definition of the functions but explicit in the statement of the lemmas (line 9 and 13). Note that the type of the arguments may be inferred by Coq. In the case of the second lemma (line 13), the only type annotation is the type of `f`: this annotation is necessary for Coq to infer the types of the other arguments. Lines 14–20 are a proofs script written with tactics. When the system reaches the **Qed** command it type-checks the obtained proof term with respect to the statement of the lemma (which is a type).

```
Fixpoint map_aux `(f: A⟶B)(l: list A)( acc: list B): list B :=
  match l with
  | [] ⇒acc
  | head:: tail ⇒map_aux f tail (( f head):: acc)
  end.

Definition map' `(f: A⟶B) l := List.rev' (map_aux f l []).

Lemma map_aux_acc: ∀ (A B: Type) (f: A⟶B) l acc,
  map_aux f l acc = (map_aux f l []) ++ acc.
Proof. (* omitted *) Qed.

Lemma equiv: ∀ A B (f: A⟶B) l, map f l = map' f l .
Proof.
 intros A B f; induction l as [ | head tail IH ].
 - trivial.
 - unfold map', rev' in *. simpl.
   rewrite map_aux_acc, ←rev_alt, rev_app_distr.
   simpl. f_equal. now rewrite IH, rev_alt.
Qed.
```

**Figure 1: A Coq example**

In addition to the features briefly presented here, SyDPaCC relies on Coq's module system [24] and type-classes. Basically, a module is an encapsulation mechanism just used to put together definitions and lemmas, theorems, *etc.*. Type modules can be thought of as module interfaces. Modules can be parametric and take modules as arguments. In this case the argument should be given a module type. Coq's module system is similar to OCaml's module system.

Type-classes are essentially definitions of record types and values of these record types are called *instances*. The difference to record types (which also exist in Coq) is that instances are stored in a database. Instances may depend on other instances, meaning one needs to have the argument instances in order to be able to build an instance that depends on them. This can be thought of as Prolog rules and instances without arguments can be thought of as Prolog facts. A Prolog-like resolution mechanism indeed exists and computes instances when one or several arguments whose type are type-classes are declared *implicit*. When such a function is called, Coq tries to build the missing implicit arguments via Prolog-like resolution. Unlike the type-classes of Haskell, there may be several possible available instances in the context where one is needed. For rules, a notion of priority is used and can be modified by the user, for facts the last defined instance is chosen.

Coq provides an extraction mechanism [27]. Coq extraction converts verified Coq definitions into compilable code in functional programming languages like OCaml, Haskell, and Scheme. It involves translating to a mini-ML language, which is then converted to the desired programming language. Non-computational parts are removed during extraction, leaving only the essential components in the intermediate representation.

## 2.2 Apache Spark

Apache Spark [42] is a framework for data analytics that focuses on querying or manipulating extensive amounts of data. The programming interface of Spark reflects a functional/higher-order programming model. This characteristic facilitates a direct and reliable extraction of parallel code from functional Coq specifications.

```
1 // Create a SparkSession
2 val sc = SparkSession.builder.appName("Example")
3                        .getOrCreate().sparkContext
4 val numbers = sc.parallelize(List(1, 2, 3, 4, 5))
5 // Use map and fold to calculate the sum of the numbers
6 val sumResult = numbers.map(_ * 2).fold(0)(_ + _)
7 // Use map and count to count the number of even numbers
8 val evenCount = numbers.map(_ % 2).count()
9 spark.stop()
```

**Figure 2: A Spark Example**

Spark advocates a data-parallel approach, which involves utilizing a dedicated distributed data structure known as RDDs (Resilient Distributed Datasets) that are partitioned by the master node. These partitions are distributed among the worker nodes along with tasks that need to be executed on RDDs. In Spark, a task refers to a series of operations carried out on an RDD. A Spark job consists of a stage, grouping a set of tasks, to operate for a job.

The operations are categorized into two categories of functions. First, transformations are applied to RDDs to create a new RDD (e.g., `map`, `filter`, `flatMap`). Transformations are lazy, meaning they do not execute immediately but instead create a lineage of transformations to be executed later. On the other hand, actions in Spark are operations that trigger the execution of the transformations and return a result or perform a specific action (e.g., `collect`, `reduce`, `fold`). Contrary to transformations, actions are eager, meaning they cause the execution of the transformations and return a value. This mechanism allows the optimization of a Spark stage by reducing computing phases and communications. Due to Spark's utilization of functions that implement functional computational patterns, programming in Spark can be considered as a skeletal approach.

Figure 2 illustrates the use of Spark high-order functions. In this example, we initialize a `SparkSession`, and create an RDD called `numbers` with a list of numbers (lines 1–4). Two examples for its use are shown. First, we use `map` to double each number, and `fold` to calculate the sum of the modified numbers (line 6). The lambda function inside `fold` adds the current value with an accumulator. Second, we use `map` to transform each number into either 0 or 1 based on whether it's even or odd. Then, we use `count` to count the number of even numbers in the RDD (line 8). Finally, we stop the `SparkSession` to release the resources (line 9). In this work, we target Scala as the language to express Spark programs. The same approach could be done using Python, or Java.

## 2.3 Bulk Synchronous Parallel ML

BSML [29] is a library for the OCaml language [26] that supports Bulk Synchronous Parallelism [46]. BSML is a pure functional library. Currently, BSML is implemented on top of MPI and can be run on large HPC systems but also more modest shared memory machines. It provides a polymorphic non-nestable data structure called *parallel vectors* and a set of 4 functions to manipulate them: `mkpar`, `proj`, `apply`, `put`. `bsp_p` is an integer constant containing the number of processors of the parallel machine. We denote this value with $p$. If $\langle v_0, \ \ldots, \ v_{p-1} \rangle$ denotes a parallel vector of size $p$ (the only possible size for a parallel vector: one value per processor),

$$\begin{array}{l}
\texttt{mkpar} : (\texttt{int} \to \alpha) \to \alpha\,\texttt{par} \\
\texttt{mkpar}\ f = \langle f\,0,\ \ldots,\ f\,(p-1)\rangle \\
\texttt{proj} : \alpha\,\texttt{par} \to (\texttt{int} \to \alpha) \\
\texttt{proj}\ \langle v_0,\ \ldots,\ v_{p-1}\rangle = \texttt{function}\ 0 \to v_0 \mid \ldots \mid p-1 \to v_{p-1} \\
\texttt{apply} : (\alpha \to \beta)\texttt{par} \to \alpha\,\texttt{par} \to \beta\,\texttt{par} \\
\texttt{apply}\ \langle f_0,\ \ldots,\ f_{p-1}\rangle\ \langle v_0,\ \ldots,\ v_{p-1}\rangle = \langle f_0\,v_0,\ \ldots,\ f_{p-1}\,v_{p-1}\rangle \\
\texttt{put} : (\texttt{int} \to \alpha)\texttt{par} \to (\texttt{int} \to \alpha)\texttt{par} \\
\texttt{put}\ \langle \mathit{tosend}_0,\ \ldots,\ \mathit{tosend}_{p-1}\rangle = \langle \mathit{rcvd}_0,\ \ldots,\ \mathit{rcvd}_{p-1}\rangle \\
\quad \text{where } \forall\, \mathit{src},\ \mathit{dst}.\ 0 \le \mathit{src}, \mathit{dst} < p \Rightarrow \mathit{rcvd}_{\mathit{dst}}\ \mathit{src} = \mathit{tosend}_{\mathit{src}}\ \mathit{dst}
\end{array}$$

**Figure 3: BSML primitives**

```
type α dlist = α list Bsml.par

let map (f: α →β) (l: α dlist): β dlist =
  Bsml.apply (Bsml.mkpar(fun _→ (List.map f))) l

let processors : int list =
  Array.to_list (Array.init Bsml.bsp_p (fun i →i))

let fold (op: α →α →α) (e: α) (l: α dlist) : α =
  let seq_fold = List.fold_left op e in
  let local_folds = Bsml.apply (Bsml.mkpar(fun _→seq_fold)) l in
  let partial_folds = Bsml.proj local_folds in
  seq_fold (List.map partial_folds processors)

let count (l: α dlist) : int = fold ( + ) 0 (map (fun _ →1) l)
```

**Figure 4: A BSML example**

the semantics of the four BSML primitives can be defined as shown in Figure 3.

A BSP program is a sequence of *super-steps*, each divided in three phases: a computation phase where processors compute in parallel with local data only, a data exchange phase and a global synchronization phase. mkpar and apply need only the computation phase to evaluate. mkpar builds a parallel vector from a function. apply applies a vector of functions to a vector of values. proj and put need data exchanges hence also a global synchronization. For both proj and put, some OCaml values are considered as empty messages thus these primitives are not necessarily communication patterns where all processors communicate with all other processors. proj can be seen as the dual of mkpar but the result function is only defined on $[0, p-1]$. In a BSP super-step, the messages to be sent can be seen as a $p \times p$ matrix where the cell $i, j$ contains the message to be sent from $i$ to $j$. put transposes this matrix and in the result, the cell $j, i$ contains the message received by $j$ from $i$. The matrices are represented as parallel vector of functions.

BSML is well suited [30] to implement *algorithmic skeletons* [7] that are higher-order functions implemented in parallel and operating on distributed data structures. Spark and its RDD can also be seen as a set of algorithmic skeletons. For example, Figure 4 implements with BSML a data-structure of distributed lists and map and fold skeletons on this structure[1].

[1]The non-pretty printed version of this example can be run either using BSML distribution https://bsml-lang.github.io or BSML online: http://tesson.julien.free.fr/try-bsml/.

## 3 THE SYDPACC APPROACH

SyDPaCC is a set of libraries for the Coq interactive theorem prover. The usual scenario for using SyDPaCC is to write a specification as an inefficient, but easy to understand, sequential functional program and to prove some rather simple properties on this program. The framework can first optimize the sequential program and second automatically parallelize the more efficient sequential program.

The front-end is responsible for optimizing specifications based on transformation theorems. For example SyDPaCC provides a variant of the third homomorphism theorem [14] that states that a function that is both leftwards and rightwards (i.e. that can be both be written using the higher-order functions List.fold_left and List.fold_right) and that has a weak form of inverse is a list homomorphism. The first homomorphism theorem states that a list homomorphism can be implemented as a composition of map and reduce. The other transformation theorem available for lists is the diffusion theorem [20].

The back-end is responsible for the automated parallelization. While the mechanism at play was designed with parallelization in mind, it is actually more general and handles any program transformation based on a change of data-structure to represent items of interest. SyDPaCC provides two notions of *correspondence*:

- A type $\mathtt{T}_{\mathrm{orig}}$ corresponds to a type $\mathtt{T}_{\mathrm{new}}$ if there exists a *surjective* function join: $\mathtt{T}_{\mathrm{new}} \to \mathtt{T}_{\mathrm{orig}}$. The surjectivity condition means that any original value can be represented with the new type. $\mathtt{T}_1 \triangleleft \mathtt{T}_2$ denotes a correspondence from $\mathtt{T}_1$ to $\mathtt{T}_2$.
- A function $\mathtt{f}\colon \mathtt{T}^1_{\mathrm{orig}} \to \mathtt{T}^2_{\mathrm{orig}}$ corresponds to another function $\mathtt{f}_{\mathrm{new}}\colon \mathtt{T}^1_{\mathrm{new}} \to \mathtt{T}^2_{\mathrm{new}}$ if $\mathtt{T}^1_{\mathrm{orig}} \triangleleft \mathtt{T}^1_{\mathrm{new}}$ and $\mathtt{T}^2_{\mathrm{orig}} \triangleleft \mathtt{T}^2_{\mathrm{new}}$ and $\forall (x : \mathtt{T}^1_{\mathrm{new}}), \mathtt{join}^2(\mathtt{f}_{\mathrm{new}}\ x) = \mathtt{f}_{\mathrm{orig}}(\mathtt{join}^1\ x)$.

We established that the composition $f \circ g$ of two functions $f, g$ in correspondence with two other functions $f_n, g_n$ is in correspondence with $f_n \circ g_n$. Other similar results with type compositions (for example pairs of types) and type correspondence, but also other function compositions (such as pairing and tupling) are part of SyDPaCC. Type and function correspondences are expressed as type-classes, and the compositions as parametrized instances.

Moreover, BSML primitives are formalized in Coq [44]. We have instances for the correspondence of Coq versions of $\alpha$ **list** with $\alpha$ dlist, and List.map with map of Figure 4.

With all these type-classes and instances, one can write:

```
Definition parallel `(f: A→B)
   `{ACorr : TypeCorr A Ap join_A} `{BCorr : TypeCorr B Bp join_B}
   `{fCorr : @FunCorr A Ap join_A ACorr B Bp join_B BCorr f fp} :
  Ap →Bp := fp.
```

At first sight, *parallel* simply returns a sub-part of one of its arguments, but note that all the formal parameters of parallel that are enclosed by curly brackets are *implicit* arguments. For a specification $\mathtt{f}_{\mathrm{spec}}$, one calls parallel $\mathtt{f}_{\mathrm{spec}}$. The type-class mechanism of Coq tries to build the missing arguments using a Prolog-like resolution. If there are type correspondences and function correspondences (which may be indirect as we have a parametrized instance for function composition) then fp is automatically built, hence $\mathtt{f}_{\mathrm{spec}}$ is automatically optimized and parallelized.

Let us consider, as a simple example of development, a generalization of the evenCount example of Figure 2 where we count the number of elements of a list that satisfy a given predicate. A possible specification for this problem follows:

```
Definition spec (p: A→bool)(l:list A) := List.length (List.filter p l).
```

where `List.filter p l` filters out the elements of list `l` that do not satisfy predicate `p`.

If we can prove that `spec` is rightwards, leftwards and has a right weak inverse, then it is a homomorphism, and it can be implemented as a composition of `map` and `reduce`.

Indeed `spec` can be written as a call to `List.fold_left` and a call to `List.fold_right` with the following binary operators:

```
Definition opl p (a: A)( count: nat) := count + if (p a) then 1 else 0.
Definition opr p (count: nat)( a: A) := count + if (p a) then 1 else 0.
```

The proofs of instances stating that `spec` is leftwards and rightwards are very short (4 and 6 lines). It is quite easy to find a right inverse for `spec` with the additional condition that there exists at least one value, named `default`, that satisfies predicate `p`:

```
Definition inv (n: nat): list A := List.map (fun _⇒default)(seq 0 n).
```

With the BSML back-end, obtaining the sequential optimization of `spec` then its parallelization is simply written:

```
Definition par_count (p: A→bool): par(list A) →nat :=
  Eval sydpacc in parallel(spec p).
```

where `Eval sydpacc` forces the immediate application of `parallel` hence producing the optimized parallel program. This kind of program is developed inside a parametrized module that takes as argument a module of type `BSML`, i.e. the module type that formalizes the BSML primitives depicted in Figure 3. The OCaml code extracted from Coq is also a parametrized module. To be able to execute the `par_count` BSML function, the user needs to apply the parametrized module to a module implementing *in parallel* the primitives of BSML. The BSML library for OCaml provides almost such a module: in the Coq formalization, processor identifiers are represented by values of type `N`, unbounded natural numbers, while in the OCaml BSML module they are represented by type `int`. Therefore SyDPaCC provides a OCaml wrapper module around the BSML OCaml `BSML` module and it is this wrapper that is passed as argument to the application parametrized module.

The Spark back-end is mostly the same, the difference being that the distributed data-structure is not `par(list A)` but `RDD A`:

```
Definition par_count (p: A→bool): RDD A → nat :=
  Eval sydpacc in parallel(spec p).
```

## 4 SYDPACC FOR SPARK: HOMOMORPHISMS

A Spark back-end for SyDPaCC enables the efficient parallelization of optimized sequential programs obtained with SyDPaCC on a Spark cluster. Spark's distributed computing capabilities provide scalability and fault tolerance, resulting in potentially enhanced performance and speed compared to the original sequential version. Just like the BSML back-end, for a Spark back-end we need:

(1) a formalization of Spark data structure and primitives,
(2) a set of type and function correspondences to relate Coq lists and functions to Spark data structure and primitives,
(3) if necessary, a Scala and Spark library to alleviate any mismatch between the formalization and the implementation,
(4) an extraction mechanism for Scala code.

We consider each of these requirements in this section.

```
1  Module Type SPARK.
2
3   Parameter RDD: Type →Type.
4   Parameter size : ∀{A: Type}, RDD A →nat.
5   Parameter get: ∀{A: Type}, RDD A →nat →A.
6
7   Definition to_list {A: Type} (d: RDD A) : list A :=
8    List.map (get d) (indexes (size d)).
9
10  Section Primitives.
11   Variable (A: Type).
12
13   Parameter parallelize: list A →RDD A.
14   Axiom parallelize_length:
15    ∀(l: list A), size(parallelize l) = length l.
16   Axiom parallelize_spec:
17    ∀(l: list A) (i: {n: nat | n < length l}),
18     get (parallelize l) (proj1_sig i) = Sig.nth l i.
19
20   Parameter map: ∀{B: Type} (f: A→B)(d: RDD A), RDD B.
21   Axiom map_length: ∀{B: Type}( f: A→B)(d:RDD A),
22    size(map f d) = size d.
23   Axiom map_spec: ∀{B: Type}( f: A→B)(d:RDD A),
24    ∀i, i < (size d) →get (map f d) i = f(get d i).
25
26   Parameter reduce: ∀`(op: A→A→A)`{Monoid A op e}( d: RDD A), A.
27   Axiom reduce_spec: ∀`(op: A→A→A)`{Monoid A op e}( d: RDD A),
28    reduce op d = Bmf.reduce op (to_list d).
29
30  End Primitives.
31
32 End SPARK
```

**Figure 5: The Coq formalization of RDDs & primitives**

*A Coq formalization of a subset of Spark.* Figure 5 presents a specification of distributed computation on Spark data structures. Line 3 introduces the `RDD` type, which stands for Spark RDDs data. The RDD type takes a type parameter, which represents the type of elements stored in the dataset. It is possible to obtain the `size` and access an element of a RDD with `get` *in the specifications*. Of course, any code that is supposed to be extracted does not use these two functions.

Lines 13–18 introduces the `parallelize` Spark operation as a function that takes a `list A` and returns a `RDD A`. `parallelize` aims at converting a regular list into an RDD for parallel processing. The `parallelize_spec` axiom specifies its behavior.

Lines 20–24 specify the behavior of `map` on RDDs. Basically, for each element in the RDD it applies the function to the element. Similarly, Lines 26–28 give a specification for the `reduce` operation. The presented axiom `reduce_spec` states that reducing an input RDD `d` with an associative operator `op`, forming a `Monoid` with its identity element `e`, should be equal to reducing (with the SyDPaCC defined `Bmf.reduce` function) the list obtained by converting `d` into a regular list using the `to_list` function.

*Type and functions correspondences.* We established the type correspondence `list A ◂ RDD A` and that the Spark `map` and Spark `reduce` presented in Figure 5 respectively correspond to `List.map` and `Bmf.reduce`. With these correspondences the sequential optimization and parallelization is available for Apache Spark.

```
import scala.of.coq.lang._
// ...
object SparkCount {
  def par_count[A](p: A => Boolean): RDD[A => Nat] =
    compose(reduce(plus)(0))
           (map(a => if (p(a)) 1 else 0))
}
```

**Figure 6: Hand-written extraction of the count example**

*A Scala library for Spark in SyDPaCC.* In Apache Spark, the definition of `map` and `fold` operations on RDDs do not need an additional implementation. To perform transformations and aggregations, the core of Spark natively provides corresponding *methods*. However methods are not functions. Therefore, we provide a wrapper library that exposes these methods as higher-order functions on RDDs.

*Limits of Coq extraction into Scala code.* Scallina [3, 11, 12], is a Coq to Scala translator. Scallina presents a grammar that encompasses a subset of both Coq and Scala, accompanied by an optimized translation approach designed specifically for programs that adhere to this grammar. The goal is to obtain very readable Scala code but at the cost of considering only a subset of Coq. To attain its goal, it does not rely on Coq's extraction to mini-ML. In particular, Scallina does not support type-classes and only supports modularity via records, not modules. As type-classes are essential to our optimization and parallelization mechanisms and that we also rely on parametrized module, Scallina cannot be used directly.

At the moment the process of Scala code generation is not automatic. One way to proceed is to look at the extracted code in OCaml and reimplement what is obtained in Coq. For the example presented in Section 4 the obtained OCaml program can be re-implemented in the Coq subset that is handled by Scallina. We then obtain the Scala+Spark code in Figure 6. This is also possible with other SyDPaCC examples. It may be possible to automate this process by using `coq-of-ocaml`[2] and then from the obtained Coq code by using Scallina. `coq-of-ocaml` translates a quite large subset of OCaml into Coq. However, in some cases, the Coq extraction uses some undocumented OCaml typing tricks that in general can be unsafe, but are not in the context of extraction. `coq-of-ocaml` does not support these features. One of our applications generates such code, thus automation would not be possible.

As our goal is not to obtain readable code, we are considering the design of an extraction mechanism to Scala embedded into Coq. However, the type system of Scala is more different from the Coq type system than OCaml and Haskell type systems.

## 5 SYDPACC FOR SPARK: DIFFUSION

By leveraging the diffusion theorem [20] and the associated accumulate skeleton, efficient programs can be computed for general accumulative computations and effectively parallelized. This allows considering a wider range a functions than just list homomorphisms.

The accumulate skeleton, as defined in [31], relies on other skeletons, including scan. The purpose of implementing the scan skeleton in Spark is thus to provide a Spark backend for the accumulate skeleton. While the current paper presents a single implementation for the scan pattern, other works propose additional implementations [33]. Exploring the feasibility and performance of these approaches when implemented on Spark is a topic for future research.

```
1  def scan[A: ClassTag](monoid: Monoid[A], e: A, xss: RDD[A])
2  : RDD[A] = {
3    val (op, i_op) = monoid.tuple()
4    val ys = xss.mapPartitions(
5                  it => Iterator(it.foldLeft(i_op)(op))
6              ).collect() // local reduce
7    val zs: Array[A] = ys.scanLeft(e)(op) // global scan
8    xss.mapPartitionsWithIndex(
9      (idx, it) => it.scanLeft(zs(idx))(op)
10   ) // local scan
11 }
```

**Figure 7: Spark implementation for the Scan skeleton**

The `scan` method does not exist on RDD. Figure 7 proposes a possible implementation to run `scan` distributively on RDDs. It is a common implementation of a known algorithm often used on distributed-memory environments. The scan function takes three parameters: `monoid`, `e`, and `xss`. The `monoid` parameter represents a type with an associative binary operation (`op`) and an identity element (`i_op`). The `e` parameter is the initial value for the scan, and `xss` is the RDD on which the scan operation will be performed. The computation consists of three phases:

(1) A local reduction of each partition (line 4–6): The input is mapped using the `mapPartitions` transformation. Each partition of data is processed by folding its elements using the `monoid` parameter. The result of each partition is collected into the driver program using the `collect` action.
(2) A global scan from the reduced values (line 7): The `scanLeft` method is applied to get the cumulative scan values.
(3) A local scan (line 8–10): Finally, each partition is processed by performing a local scan on its elements, using as starter the corresponding value from the previous phase (using the `mapPartitionsWithIndex` transformation).

To match with SyDPaCC definitions of `scan`, we also provide an implementation of `scan` on Spark that returns a pair containing both the scanned RDD, and the total reduction of the input RDD, corresponding to the $(n+1)^{\text{th}}$ value when the scan operation returns a list starting by the unit element of the monoid. To calculate this value, we simply return the value at the last index of `zs`.

The implementation of scan on Spark including tests is about 1300 LoC and 2 days human effort. All the sources of this implementation on Spark are available on a git repository[3].

The initial experimental results of the scan skeleton execution, implemented in Fig. 7, demonstrate an encouraging scalability.

We recorded the computation time of the application of `scan` on a list of $10 \times 10$ randomly generated matrices, using the monoid $(\otimes, I_{10})$ where $\otimes$ is the product of matrices, and $I_{10}$ the identity matrix of dimension 10. The experiments have been conducted first on a list of 100k elements ($L1$), and 1M elements ($L2$). The run on $L1$ took 8.85 seconds using 1 single Spark worker, 5.18 seconds with 2

[2]https://formal.land/docs/coq-of-ocaml

[3]https://anonymous.4open.science/r/SyDPaCC-Spark/sydpacc-spark

workers, and 3.33 seconds with 4 workers. Similarly, the execution of `scan` on $L2$ took 82.93s on 1 worker, 55.02s on 2 workers, and 46.02s on 4 workers.

The presented results suggests that increasing the input size has a significant impact on the computation time, leading to longer execution times. However, by employing parallelization with a higher number of workers, the execution time can be reduced. The specific performance characteristics may vary depending on the nature of the computation and the underlying algorithms used.

## 6 RELATED WORK

The skeletal approach is a key concept for reasoning on parallel programs [1], but is not a new research area. In the late 90s, many works have focused on the formal derivation of functions on linear data-structures (e.g., list, array) into efficient parallel programs. In [20] defined and used the diffusion theorem to decompose a recursive definition into several functions such that each function can be described by skeletons. Similarly, lists homomorphisms have been used to obtain programs that can be decomposed onto a composition of parallel primitives [8, 19]. In [16], Gorlatch et. al. used list homomorphisms for expressing a parallel scan operation. Dosch et. al. used the same approach for both accumulation and indexing operations on distributed lists in [10]. However all these works are based on pen and paper proofs. In more recent work, researches have used proof assistant to prove the semantics preservation of parallel programs. For example, Grégoire and Chlipala provide a small parallel language and its semantics and proves correct optimizations of stencil-based computations [17]. In [9], Daum formalized a subset of Data Parallel C using the Isabelle/HOL proof assistant.

Philippe et al. presented a formalized model transformation engine in Coq in [39]. They further extracted this engine to generate a running Spark program in two steps: first, generating sequential Scala code that corresponds to the Coq specification, and then distributing the execution by replacing Scala's sequential data structures with Spark RDDs. However, their approach has two notable weaknesses. Firstly, the Coq formalization does not include the distribution process of Spark; it only focuses on Scala functions. While the semantics of Spark functions can be considered similar to those of Scala, this limitation hinders reasoning about distributivity. Second, the extraction process was performed manually. It lacked automation and formal proof of correctness, leaving room for potential errors or inconsistencies.

The study of frameworks like Hadoop MapReduce [23, 33] and Apache Spark [6] from a functional programming perspective is relevant to our approach in SyDPaCC. Their usage of skeletons, either modeled as high-order functions or as parametrized classes, enable us to adopt a similar methodology for extracting MapReduce or Spark programs from Coq. In [37], Ono et al. verified MapReduce programs using Coq specifications, either by extracting Haskell code for Hadoop Streaming or directly writing Java programs with JML annotations. For the later, they used Krakatoa [13] for Coq lemma generation. However, their work was less automated and systematic compared to SyDPaCC.

In [21], Huisman et. al. proposed the verification of annoted program transformations. Their work targets GPU programs obtained from sequential loops, then optimized thanks to additional transformations (e.g., barrier introduction, change on data location). Thanks to this approach, they verified an application of the scan pattern to solve the maximum prefix sum (MPS) problem [41]. Their approach have been generalized for verifying CPU parallel programs [5]. Contrary to our approach, they do not aim at verifying properties expressed as annotations, but they infer from them correct new properties on target programs expressed as kernel loops. Also, they target specific programs, decreasing the scope of the results (e.g., they could verify scan pattern and solve the MPS problem using it).

## 7 CONCLUSION AND FUTURE WORK

Apache Spark is widely popular in the field of big data processing due to its powerful capabilities, flexible architecture, and the large spectrum of applications it covers (e.g, graph processing with GraphX [15]). Targeting Spark as a back-end solution for SyDPaCC opens up new possibilities for applying formal verification techniques to big data processing. It opens the possibility of the verification of algorithms, data transformations, and machine learning models implemented in Spark, providing increased confidence in the correctness and reliability of these systems.

In this paper, we presented ongoing work towards an extension of SyDPaCC to extract verified Scala code for running Spark programs from a Coq specification. In future work, we plan to conduct experiments on a distributed architecture to estimate the scalability of the execution of SyDPaCC skeletons on top of Apache Spark. Moreover, we plan to extend the set of functions on RDDs to provide support for all the SyDPaCC specification including non-linear data structures like binary trees [38]. Of course our priority is to be able to automatically extract Scala code from Coq to be able to easily use the extracted code in Apache Spark applications.

Spark provides a wide range of configuration options that affect various aspects of its execution, such as memory allocation, parallelism settings, task scheduling, and data serialization. These configuration parameters can greatly influence the performance and behavior of Spark applications. However, formalizing such aspects can be challenging due to the dynamic and highly configurable nature of Spark [28]. It is a non-trivial task that requires careful consideration of the system's complexity and dynamic behavior. In addition, Spark code is run on top of the Java Virtual Machine (JVM), which processes optimization while executing. Modeling performance characteristics of applications running on the JVM is more challenging compared to languages like OCaml when considering algorithm ran with BSML. OCaml has a more predictable performance model due to their ahead-of-time compilation and relatively simpler runtime system. In contrast, because of the JVM dynamic optimizations (e.g., garbage collection, just-in-time compilation), it is difficult to accurately predict the performance of a Spark application.

## ACKNOWLEDGMENTS

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations (see https://www.grid5000.fr).

## REFERENCES

[1] M. Aldinucci and M. Danelutto. 2007. Skeleton-based parallel programming: Functional and parallel semantics in a single shot. *Comput Lang Syst Str* 33, 3-4 (2007), 179–192.

[2] Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, and Matei Zaharia. 2015. Scaling Spark in the Real World: Performance and Usability. *PVLDB* 8, 12 (2015), 1840–1851. http://www.vldb.org/pvldb/vol8/p1840-armbrust.pdf

[3] Youssef El Bakouny and Dani Mezher. 2018. The Scallina Grammar - Towards a Scala Extraction for Coq. In *Formal Methods: Foundations and Applications (SBMF) (LNCS, Vol. 11254)*. Springer, 90–108. https://doi.org/10.1007/978-3-030-03044-5_7

[4] Y. Bertot and P. Castéran. 2004. *Interactive Theorem Proving and Program Development*. Springer. https://doi.org/10.1007/978-3-662-07964-5

[5] S. Blom, Saeed Darabi, Marieke Huisman, and M. Safari. 2021. Correct program parallelisations. *International Journal on Software Tools for Technology Transfer* 23 (10 2021), 1–23. https://doi.org/10.1007/s10009-020-00601-z

[6] Yu-Fang Chen, Chih-Duo Hong, Ondrej Lengál, Shin-Cheng Mu, Nishant Sinha, and Bow-Yaw Wang. 2017. An Executable Sequential Specification for Spark Aggregation. In *Networked Systems (NETSYS)* (Marrakech, Morocco) *(LNCS, Vol. 10299)*. 421–438. https://doi.org/10.1007/978-3-319-59647-1_31

[7] Murray Cole. 1989. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press.

[8] Murray Cole. 1995. Parallel Programming with List Homomorphisms. *Parallel Processing Letters* 5, 2 (1995), 191–203.

[9] M. Daum. 2007. Reasoning on Data-Parallel Programs in Isabelle/Hol. In *C/C++ Verification Workshop*. https://doi.org/~rhuuck/CV07/program.html

[10] W. Dosch and B. Wiedemann. 2000. List Homomorphisms with Accumulation and Indexing. In *Trends in Functional Programming*, G. Michaelson, P. Trinder, and H.-W. Loidl (Eds.). Intellect, 134–142.

[11] Youssef El Bakouny, Tristan Crolard, and Dani Mezher. 2017. A Coq-based synthesis of Scala programs which are correct-by-construction. In *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, Barcelona, Spain, June 20, 2017*. ACM, 4:1–4:2. https://doi.org/10.1145/3103111.3104041

[12] Youssef El Bakouny and Dani Mezher. 2018. Scallina: Translating Verified Programs from Coq to Scala. In *Asian Symposium on Programming Languages and Systems (APLAS) (LNCS, Vol. 11275)*. Springer, 131–145. https://doi.org/10.1007/978-3-030-02768-1_7

[13] Jean-Christophe Filliâtre and Claude Marché. 2007. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *19th International Conference on Computer Aided Verification (LNCS)*, W. Damm and H. Hermanns (Eds.). Springer.

[14] J. Gibbons. 1996. The third homomorphism theorem. *J Funct Program* 6, 4 (1996), 657–665. https://doi.org/10.1017/S0956796800001908

[15] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 599–613.

[16] S. Gorlatch. 1996. Systematic Efficient Parallelization of Scan and Other List Homomorphisms. In *Euro-Par'96. Parallel Processing (LNCS, 1123–1124)*, L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert (Eds.). LIP-ENSL, Springer, Lyon.

[17] Thomas Grégoire and Adam Chlipala. 2016. Mostly Automated Formal Verification of Loop Dependencies with Applications to Distributed Stencil Algorithms. In *Interactive Theorem Proving (ITP) (LNCS, Vol. 9807)*, Jasmin Christian Blanchette and Stephan Merz (Eds.). Springer, 167–183. https://doi.org/10.1007/978-3-319-43144-4_11

[18] William A. Howard. 1980. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin and J. R. Hindley (Eds.). Academic Press, 479–490.

[19] Zhenjiang Hu, Hidewaki Iwasaki, and Masato Takeichi. 1997. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Trans Program Lang Syst* 19, 3 (1997), 444–461. https://doi.org/10.1145/256167.256201

[20] Z. Hu, M. Takeichi, and H. Iwasaki. 1999. Diffusion: Calculating Efficient Parallel Programs. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*. ACM, 85–94.

[21] Marieke Huisman, Stefan Blom, Saeed Darabi, and Mohsen Safari. 2018. Program Correctness by Transformation. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I* (Limassol, Cyprus). Springer-Verlag, Berlin, Heidelberg, 365–80. https://doi.org/10.1007/978-3-030-03418-4_22

[22] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. https://doi.org/10.1145/1743546.1743574

[23] Ralf Lämmel. 2008. Google's MapReduce programming model – Revisited. *Sci Comput Program* 70, 1 (2008), 1–30. https://doi.org/10.1016/j.scico.2007.07.001

[24] Xavier Leroy. 2000. A modular module system. *J Funct Program* 10, 3 (2000), 269–303. https://doi.org/10.1017/S0956796800003683

[25] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[26] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2022. The OCaml System release 5.00. https://v2.ocaml.org/manual/.

[27] Pierre Letouzey. 2008. Coq Extraction, an Overview. In *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008 (LNCS, Vol. 5028)*, A. Beckmann, C. Dimitracopoulos, and B. Löwe (Eds.). Springer. https://doi.org/10.1007/978-3-540-69407-6_39

[28] Chen Lin, Junqing Zhuang, Jiadong Feng, Hui Li, Xuanhe Zhou, and Guoliang Li. 2022. Adaptive Code Learning for Spark Configuration Tuning. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1995–2007. https://doi.org/10.1109/ICDE53745.2022.00195

[29] Frédéric Loulergue. 2017. A BSPlib-style API for Bulk Synchronous Parallel ML. *Scalable Computing: Practice and Experience* 18 (2017), 261–274. Issue 3. https://doi.org/10.12694/scpe.v18i3.1306

[30] Frédéric Loulergue. 2017. Implementing Algorithmic Skeletons with Bulk Synchronous Parallel ML. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 461–468. https://doi.org/10.1109/PDCAT.2017.00079

[31] Frédéric Loulergue. 2017. A Verified Accumulate Algorithmic Skeleton. In *Fifth International Symposium on Computing and Networking (CANDAR)*. IEEE, Aomori, Japan, 420–426. https://doi.org/10.1109/CANDAR.2017.108

[32] Frédéric Loulergue, Wadoud Bousdira, and Julien Tesson. 2017. Calculating Parallel Programs in Coq using List Homomorphisms. *Int J Parallel Prog* 45 (2017), 300–319. Issue 2. https://doi.org/10.1007/s10766-016-0415-8

[33] Kiminori Matsuzaki. 2016. Functional Models of Hadoop MapReduce with Application to Scan. *Int J Parallel Prog* (2016). https://doi.org/10.1007/s10766-016-0414-9

[34] Toby C. Murray and Paul C. van Oorschot. 2018. BP: Formal Proofs, the Fine Print and Side Effects. In *2018 IEEE Cybersecurity Development (SecDev)*. IEEE Computer Society, Cambridge, MA, USA, 1–10. https://doi.org/10.1109/SecDev.2018.00009

[35] T. Nipkow, L. C. Paulson, and M. Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer.

[36] Martin Odersky, Lex Spoon, and Bill Venners. 2010. *Programming in Scala* (second ed.). Artima.

[37] Kosuke Ono, Yoichi Hirai, Yoshinori Tanabe, Natsuko Noda, and Masami Hagiya. 2011. Using Coq in specification and program extraction of Hadoop MapReduce applications. In *SEFM (LNCS)*. Springer, Berlin, Heidelberg, 350–365. https://doi.org/10.1007/978-3-642-24690-6_24

[38] Jolan Philippe and Frédéric Loulergue. 2019. Parallel Programming with Coq: Map and Reduce Skeletons on Trees. In *ACM Symposium on Applied Computing (SAC)*. ACM, 1578–1581. https://doi.org/10.1145/3297280.3299742

[39] Jolan Philippe, Massimo Tisi, Hélène Coullon, and Gerson Sunyé. 2021. Executing Certified Model Transformations on Apache Spark. In *14th ACM SIGPLAN International Conference on Software Language Engineering (SLE)* (Chicago, IL, USA). ACM, New York, NY, USA, 36–48. https://doi.org/10.1145/3486608.3486901

[40] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2023. *Logical Foundations*. Software Foundations, Vol. 1. Electronic textbook. http://softwarefoundations.cis.upenn.edu Version 6.3.

[41] Mohsen Safari, Wytse Oortwijn, Sebastiaan Joosten, and Marieke Huisman. 2020. Formal Verification of Parallel Prefix Sum. In *NASA Formal Methods: 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11–15, 2020, Proceedings* (Moffett Field, CA, USA). Springer-Verlag, Berlin, Heidelberg, 170–186. https://doi.org/10.1007/978-3-030-55754-6_10

[42] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. 2016. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics* 1 (2016), 145–164.

[43] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019). https://doi.org/10.1017/S0956796818000229

[44] Julien Tesson and Frédéric Loulergue. 2011. A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation. In *International Conference on Computational Science (ICCS)*. Elsevier, Singapore, 36–45. https://doi.org/10.1016/j.procs.2011.04.005

[45] The Coq Development Team. [n. d.]. The Coq Proof Assistant. http://coq.inria.fr.

[46] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103. https://doi.org/10.1145/79173.79181

[47] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Received 2023-05-26; accepted 2023-06-23