SyDPaCC: A Framework for the Development of Verified Scalable Parallel Functional Programs

Frédéric Loulergue, Jordan Ischard

PREPRINT

SyDPaCC: A Framework for the Development of Verified Scalable Parallel Functional Programs

Frédéric Loulergue Jordan Ischard

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

Abstract: The SyDPaCC framework supports the development of scalable parallel functional programs with Coq and helps the developers to write correct-by-construction programs with respect to specifications written as simple (and possibly very inefficient) functional programs. Parallel programs are built from specifications using verified program transformations offered by SyDPaCC. Leveraging the Coq extraction mechanism, compilable code can be obtained and executed on shared-memory or large scale distributed memory parallel machines. This paper presents the usage of SyDPaCC via an example, explains the internals of SyDPaCC and gives a tour of the program transformations provided by the framework.

Keywords: scalable parallel computing, functional programming, interactive theorem proving, program transformation, Coq

1 Introduction

There are many ways to program parallel architectures ranging from a dozen cores in a shared memory machine to tens of thousands of cores or more in large distributed memory supercomputers. Message Passing Interface [72, 61] is a de facto standard. While it is high-level compared to low-level shared memory and network APIs, it remains low-level compared to other approaches. In particular, MPI follows the Single Program Multiple Data (SPMD) paradigm in which programmers write a single program (implicitly parametrized by a process identifier) that should be understood as a parallel composition of communicating sequential programs. This style helps writing programs that scale to large numbers of processors and gives developers a lot of control on the parallel aspects of their programs. However, it is difficult for the programmers to see if a piece of code is *local* because it depends on the process identifier or if it is *global* because it is independent of the process identifier. Moreover, the execution and reading order may be different.

Automatic parallelization, in particular of nested loops using the polyhedral model [23, 8, 69], does not require any expertise in parallelism. It is however mostly used for automatic vectorization and parallelization for shared memory architectures, thus with a limited scalability. The programmers also completely lose control over parallelism, although recent approaches aim at making polyhedral optimizations amenable to programmers' analysis and customization [2].

There are intermediate approaches where the programmers still have some control over the parallel aspects of their programs but in a more restrictive parallel model than the parallel model supported

by MPI. The parallel model may be restricted independently of any programming interface, for e.g. the PRAM model [42], the Bulk Synchronous Parallel model [82], or the logP model [13]. Or the restricted parallel model may come from a high-level programming language or library which allows expressing, without complex implementation details, a large range of parallel algorithms.

Functional programming languages have been an inspiration for the design of such high-level parallel programming languages and libraries, starting with algorithmic skeletons [11, 65, 70, 31]. Such skeletons can be seen both as patterns of parallel algorithms and higher-order functions implemented in parallel and working on concurrent or distributed data structures. There are many algorithmic skeletons libraries for various programming languages in particular C++ [21, 22, 35, 46]. Popular frameworks such as MapReduce [15] and Apache Spark [1] are closely related to algorithmic skeletons. Other proposals like Bulk Synchronous Parallel ML [56] (BSML) and Eden [49] and are more general than data-parallel skeletons and can be used to implement them [43, 51].

Gallina, the specification language of the Coq [81, 3, 10] proof assistant can be considered as a functional programming language with a rich type system allowing to state and prove mathematical properties. Based on the Curry-Howard correspondence [37], statements of properties are expressed as types while the proofs are programs, even though one usually writes proofs in Coq using one of the proof script languages (in particular Ltac) rather than directly writing programs that are proofs.

The starting point of the approach presented in this paper was to answer the question : is it possible to reason about programs written with one of these high-level library, namely BSML, using Coq and following a shallow embedding approach, i.e. using Coq as a sequential functional programming language? This is indeed the case [26, 80] as BSML has a pure functional semantics [55]. Note that BSML is *axiomatized* in Coq, therefore BSML programs cannot be executed *in parallel* in Coq. But they can be extracted from Coq to parametrized OCaml code which can be instantiated by a parallel implementation of BSML on top of MPI [56] and run on parallel machines. However, we also provide a *sequential* realization of the BSML axioms in Coq which allows running *sequentially* BSML programs in Coq. It also allows validating experimentally that the parallel BSML implementation returns the same results as the sequential Coq implementation which is proved correct with respect to the BSML axiomatized semantics.

Algorithmic skeletons can be implemented using BSML, therefore Coq and BSML axioms can be leveraged to verify the correctness of implementations of algorithmic skeletons [29, 50]. It is convenient to express the correctness of an algorithmic skeleton or a non-skeletal parallel program written in BSML by writing a sequential version that is considered as a specification and by stating that they compute the same results for the same inputs (up to a sequentialization of distributed data structures).

Constructive algorithmics [5] is a transformational approach to optimize sequential functional programs. When applied to parallel programming [12, 32, 62], the transformations are proved (on paper) on sequential programs and the result of the transformations is a sequential program expressed with higher-order functions, such as map and reduce. The parallel implementation is then written using an algorithmic skeleton library (often in C++) using the optimized sequential program as a specification. It is of course possible to write and verify such sequential transformations in Coq, and the correctness of algorithmic skeletons BSML implementations in Coq guarantees that replacing the higher-order sequential functions by corresponding algorithmic skeletons is correct.

At some point, we realized that transformations and the correctness of algorithmic skeletons can be expressed as instances of typeclasses in Coq in a way that the typeclass instance resolution mechanism of Coq can automatically perform the sequential program optimizations and replacement of high-order sequential functions by algorithmic skeletons. We consider the replacement phase to be an automatic parallelization. This initial transformation and parallelization infrastructure was based on many typeclasses. It was later simplified and restructured into what is now called the SyDPaCC framework. This refactoring made more clear that only the statements of the correctness of algorithmic skeletons (and their implementations) depend on the parallel model. Other parallel

libraries than BSML can be axiomatized, and algorithmic skeletons implemented and proved correct using these libraries, while relying on the existing theorems for optimization. This part of SyDPaCC is called a back-end and the only fully supported back-end is BSML, but there is preliminary work on Apache Spark [53].

The SyDPaCC framework [59] thus supports the development of scalable parallel functional programs with Coq and helps the developers to write correct-by-construction programs with respect to specifications written as simple (possibly very inefficient) functional programs. Parallel programs are built from specifications using provided verified program transformations. Coq extraction mechanism [48] generates parametrized compilable code that can be executed on shared-memory or large scale distributed memory parallel machines. SyDPaCC is flexible and accessible to users with different expertises and interests.

Users knowledgeable in sequential functional programming and familiar with interactive theorem proving with Coq can write inefficient sequential programs (considered as specifications) and prove properties on these programs to automatically transform them into efficient algorithmic skeleton-based parallel programs. In this case, the framework is limited to functions that are compositions of list homomorphisms [5], accumulations [4], BSP homomorphisms [29, 45] and sequential functions that are not in these classes but for which a proof of correspondence with a parallel version exists. The paper presents SyDPaCC mostly with this kind of users in mind.

Users who also have knowledge about the bulk synchronous parallel model can write any bulk synchronous parallel algorithm in a functional way and prove its correctness using the framework. It is of course no automatic, but SyDPaCC does provide reasoning principles that help reasoning about BSML programs. Such users, by providing the correctness statement as a typeclass instance, extend the expressivity of the framework for less knowledgeable users.

After a quick introduction to Coq (Section 2), this paper presents the use of SyDPaCC via a new and very simple example (Section 3) including experiments on a parallel machine. Section 4 explains how a parallel program can be automatically obtained from the specification of Section 3 and the model of parallelism SyDPaCC currently supports. We give a tour of the program transformations available in SyDPaCC and the related publications (Section 5). Related work is discussed in Section 6. We conclude and give future research directions in Section 7.

SyDPaCC is open source software and is available at `https://sydpacc.github.io`. Code examples are (sometimes slightly simplified) excerpts of SyDPaCC version 0.5.

2 An Overview of Coq

Gallina is quite close to the pure functional syntax of OCaml, but with a richer type system. One feature not present in the OCaml functional language is the support of typeclasses [73]. Basically, typeclasses in Coq are structures or record types but that come with a kind of Prolog-like database.

We give a very quick overview of the salient features of Coq used in SyDPaCC with the code of Figure 1.

Lines 4–8, 10, 17–21 are close to usual functional programming: they contain the definitions of two recursive functions `length` and `filter` by pattern matching on their list arguments, as well as the definition of a function `compose` for which an infix notation ∘ is given in line 11.

These functions are polymorphic functions. In the case of `length` (line 4), the type argument `A` is explicitly given. However, because of the option `Set Implicit Arguments` in line 2, Coq makes this argument *implicit*. Indeed, having the type of the second argument of `length` determines `A`. That is why on line 7, `length` is applied only to one argument. `filter` is also polymorphic and depends on a type argument. Line 17 does not contain `(A: Type)` as first argument. Nevertheless, line 2 also contains the option `Generalizable All Variables` and the argument `(p: A→bool)` of `filter` is prefixed with a back-tick `‘`. It means that if the type of the argument uses names that are

```
Require Import List. Import ListNotations.
Set Implicit Arguments. Generalizable All Variables.

Fixpoint length (A: Type)(xs: list A) : nat :=
  match xs with
  | [] ⇒ 0
  | _::xs' ⇒ 1 + length xs'
  end.

Definition compose `(g: B→C)`(f: A→B): A→C := fun x ⇒ g (f x).
Infix "∘" := compose (at level 40, left associativity).

Lemma length_app: ∀ A (xs ys: list A),
  length (xs++ys) = length xs + length ys.
Proof. (* omitted *) Qed.

Fixpoint filter `(p: A→bool)(xs: list A): list A :=
  match xs with
  | [] ⇒ []
  | x::xs' ⇒ if p x then x::filter p xs' else filter p xs'
  end.

Lemma filter_app: ∀ `(p: A→bool) xs ys,
  filter p (xs++ys) = filter p xs ++ filter p ys.
Proof.
  induction xs as [ | x xs IH ].
  - easy.
  - intro ys. simpl. rewrite IH. now destruct (p x).
Qed.

Class Homomorphic `(h: list A→B)`(op: B→B→B) :=
  { homomorphic : ∀ x y, h (x++y) = op (h x) (h y) }.

Instance length_hom (A: Type): Homomorphic (@length A) Nat.add.
Proof. constructor. apply length_app. Qed.

Instance filter_hom `(p: A→bool): Homomorphic (filter p) (@List.app A).
Proof. constructor. apply filter_app. Qed.

Instance composition_homomorphic
  `{Hg: @Homomorphic A (list B) g (@List.app B)}
  `{Hf: @Homomorphic B C f op_f } : Homomorphic (f ∘ g) op_f.
Proof. (* omitted *) Qed.
```

Figure 1: A Simple Coq Example

not defined yet, these names are automatically added as implicit arguments. In the case of `filter`, an argument `(A: Type)` is added automatically because the type of `p` is `A→bool`.

`filter p xs'` (on line 24) is the application of function `filter` to two arguments: `p` and `xs'`. `++` is a notation for list concatenation. The `match with end` construct is a kind of extended `switch` similar to the *match* construct of OCaml or Rust. In line 19, it means that if the list is the empty list `[]` then the returned value is the empty list `[]` ; otherwise (line 20) the list is non-empty: it consists of a first element `x` and the remainder (called `xs'`) of the list `xs`. `::` is the "cons" operation that builds a list by adding an element in front of an existing list. This operation is also a constructor of lists, it can be used in such a *pattern*. On the right of ⇒ , variable `x` is bound to the first element of the argument list `xs` and `xs'` to `xs` but without its first argument.

Unlike mainstream functional programming languages, it is possible to state and prove lemmas in Coq. The first lemma (lines 13–15) states that the length of the concatenation of two lists is the sum of the lengths. Its proof script is omitted (line 15). The second lemma states that filtering the concatenation of two lists is the concatenation of the filtering of both lists. The proof script is given in extenso on lines 25–29. It consists of a sequence of *tactics*. Using a Coq IDE, these tactics are applied in proof mode on a goal and may produce new goals or end the proof of a goal. For example, the first tactic starts a proof by induction on the list `xs`, the initial goal being the lemma statement. This tactics produces two new goals:

- ∀ `ys: list A, filter p ([]++ys) = filter p [] ++ filter p ys`,
- ∀ `ys: list A,`

  `filter p ((x::xs)++ys) = filter p (x::xs)++filter p ys.`

The first goal can be proved by simplification using the definitions of `filter` and list concatenation, then by concluding by reflexivity. This is done automatically using the tactic `easy`. For the second goal, we need to apply the induction hypothesis (named `IH`) and reason by case on whether `x` satisfies predicate `p`. This is done by the sequence of tactics of line 28.

Informally, a function f is a list homomorphism if it can be computed by dividing the list in two pieces, recursively calling f on these two pieces and combining the two results into the final result using a binary operation. We define the *typeclass* `Homomorphic` on lines 31–32, to formalize this concept. One may wonder why `h` and `op` are not bundled in the typeclass as fields `h` and `op`. This is a quite advanced Coq technical discussion and we refer to [76]. In this paper, `h` and `op` can be considered as part of the `Homomorphic` record but having them as arguments to the typeclass allows the instance resolution mechanism briefly presented below to infer them if needed.

We then define an *instance* of this class on lines 34–35. The field `homomorphic` is the statement of a property, thus to define such an instance we need to write a proof. In this case, we rely on the lemma `length_app` already proved. The interest of typeclasses is the typeclass instance resolution mechanism. It is possible to specify arguments of a function in Coq to be implicit. If an implicit argument has for type a typeclass, then Coq uses a Prolog-like resolution mechanism to find or build an instance of this type. When we define the instance `length_hom`, this instance is added to a database of instances, and can be thought as a Prolog fact. Some instances may have parameters and then correspond to Prolog rules (see Figure 3 for examples of such instances). These facts and rules are used by the resolution mechanism of Coq.

3 SyDPaCC by Example

As a small example of the use of SyDPaCC, we will implement a parallel program for counting the number of elements in a distributed list satisfying a given predicate.

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

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

Definition count_inv (n: nat): list A :=  map (fun x⇒ default) (seq 0 n).

Instance count_leftwards: Leftwards count_spec opl 0.

Instance count_rightwards: Rightwards count_spec opr 0.

Instance count_right_inverse: Right_inverse count_spec count_inv.
```

Figure 2: Properties of `count_spec`

Specification. The specification is given as a sequential function manipulating a list. It can simply be written as:

```
Definition count_spec (A: Type)(p: A→bool)(l: list A): nat :=
  length (filter p l).
```

We omit the details here and refer to the code, but we assume that there exists at least one value of type `A` that satisfies `p`. This value is named `default`.

Properties of the Specification. We prove three properties about this specification:

- `count_spec` is rightwards, meaning it can be implemented using `fold_right`,
- `count_spec` is leftwards, meaning it can be implemented using `fold_left`,
- `count_spec` has a right inverse. For a positive number n, this right inverse builds a list containing only the value `default` n times.

Omitting the proofs of the instances (that are only a few lines long), this is done by the code of Figure 2.

Program Transformation and Parallelization. Finally, we call the program transformation and automatic parallelization of SyDPaCC on the specification:

```
Definition par_count (A: Type)(p: A→bool): par(list A)→nat :=
  parallel(count_spec p).
```

The transformations are done by the call to `parallel`. One important aspect besides applying it to `count_spec` is the fact that we expect the result of this call to be of type `par(list A)→nat`. `par` is a parametric type of distributed vectors (see more in Section 4.1). `par(list A)` is therefore a parallel vector of lists and can be thought as a distributed list. Thus, we want to obtain a function that operates on distributed lists.

This call succeeds, and `count_par` is the composition of a parallel reduce and a parallel map. `parallel` relies on implicit arguments and typeclasses as explained in Section 4.2. The transformation proceeds in two steps: (1) it transforms the specification into a composition of sequential map and reduce, (2) these sequential functions are replaced with their parallel counterparts. Note that the first transformation may decrease the algorithmic complexity of the sequential function (this is not the case here). This is for example the case for the maximum prefix sum application [59].

```
Instance third_homomorphism_theorem `{h:list A→B}
  `{Hl: Leftwards A B h opl e}
  `{Hr: Rightwards A B h opr e}
  `{inv: Right_inverse A B h h'}:
  Homomorphic h (fun l r ⇒ h( (h' l)↤(h' r)))

Instance first_homomorphism_theorem
  `{H: Homomorphic A B h op}
  FunCorr h ((reduce op) ∘ (List.map (fun x⇒ h[x]))).
```

Figure 3: Third and First Homomorphism Theorems as Typeclass Instances

In the case of the `count_spec`, a part of the call to `parallel` proceeds as follows. `count_spec` properties stated as instances in Figure 2 are the hypotheses of the third homomorphism theorem (Figure 3, proofs are omitted) and the conclusion of this theorem is stated as a typeclass instance: if these hypotheses hold then the function is homomorphic. This conclusion in turn is the hypothesis of the first homomorphism theorem, that states — as an instance of the typeclass `FunCorr` which is, in this specific application, equivalent to stating `∀ l, h l = (reduce op) (map (fun x⇒ h[x]) l)` — that when a function is homomorphic then it can be implemented as a composition of `map` and `reduce`.

Basically, `parallel` tries to find or build instances of `FunCorr` for its argument. For `count_spec`, one possible instance is `first_homomorphism_theorem`. As this theorem has an instance of `Homomorphic` as argument, then the instance resolution mechanism tries now to find an instance of `Homomorphic`. One possible way of building such an instance is to apply `third_homomorphism_theorem` which requires to build instances of `Leftwards`, `Rightwards` and `Right_inverse`. There are actually such instances already defined in Figure 2. The instance resolution succeeds. This is however not the whole story: what we have just described gives a sequential function of type `list A→nat` not of the expected type `par(list A)→nat`. Indeed, `FunCorr` is more than just extensional function equality and there are additional `FunCorr` instances that take care of replacing `map` and `reduce` by their parallel counterparts as explained Section 4.2.

Note that resolution succeeds this way and proposes this particular parallelization because there are instances of properties of `count_spec` as shown in Figure 2. Proving different properties could instead trigger the transformation and parallelization using other transformation theorems and algorithmic skeletons. Also, it is not difficult to directly show that `count_spec` is indeed homomorphic by writing an instance of `Homomorphic`. In this case, the third homomorphism theorem would not be applied, but the obtained parallel program would be very similar: the arguments to the parallel reduce and the parallel map, although extensionally equal to what is obtained here, may be implemented differently.

Code Extraction and Experiment. The last step is to obtain OCaml code from this Coq development using the extraction mechanism. All the applications in SyDPaCC are written inside a parametric module that takes as argument a module whose signature is the module type of BSML primitives. To obtain executable code, we just need to apply this parametric module to an actual implementation of BSML primitives as provided by the BSML [56] library for OCaml implemented on top of an MPI C library.

We experimented on a parallel machine named Speed2: a 32-core processor (Intel Xeon Gold 5218) with 192 GB of memory, running Ubuntu 22.04, OCaml 4.13.1, GCC 11.4 and OpenMPI 4.1.2. We measured the time required for the computation 30 times. The table and figure present the median of these measurements together with the standard deviation, as well as the relative speed-up.

Table 1 and Figure 3 present the results for `count_par` where the elements are square floating-point number matrices of size 50×50 and the predicate checks if the square of the matrix is diagonal. The experiment was conducted on a list of length 2^{10}. The speed-up is less than the number of processors: indeed communications and synchronization are needed. Note that both communications and synchronization are independent of the length of the list. Therefore, increasing the length of the list would also increase the speed-up.

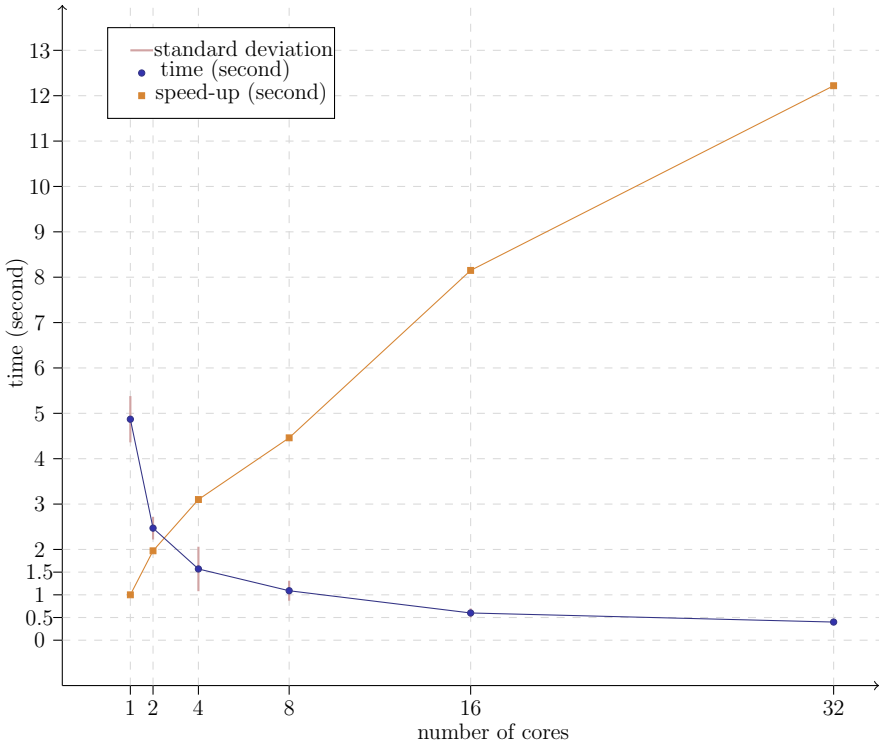

Figure 4: Experiments on a 32-core Parallel Machine

Table 1: Result table for experiments on a 32-core Parallel Machine

number of cores	median time (second)	standard deviation	speed-up
1	4.87	0.51	1
2	2.47	0.25	1.97
4	1.57	0.49	3.10
8	1.09	0.22	4.46
16	0.60	0.1	8.15
32	0.40	0.04	12.22

3.1 Trusted Base

The goal of SyDPaCC is to obtain verified parallel programs: how sure are we that the obtained programs are correct?

First, we need to trust the Calculus of Induction Constructions (CIC) on which Coq's type system is based, as well as the type checker of Gallina programs of Coq. One of the goals of the MetaCoq project [75] is to verify these aspects, as there were issues with the type checker (mostly related to checking the termination of recursive functions). A large subset of the type checker has been verified [74], thus we now only need to trust the CIC theory.

Second, the extraction mechanism of Coq have to be trusted. Actually, there are two parts in this mechanism: in a first phase, Gallina programs are extracted into an internal miniML language. This part (erasure) has been verified using MetaCoq [74]. The second phase translates miniML to mainstream functional languages such as Haskell (some Gallina features are not supported) and OCaml. This second phase is not verified, and in the case of OCaml, the extracted code may contain values and types of the `Obj` undocumented module which are used to bypass OCaml's type checker. As the code is typed in Coq, if used correctly, these tricks should be safe. In the case of `count_spec`, the extracted code contains only one such type definition which is not used by the application code. This is not the case for some applications discussed in the next section. An alternative to Coq's built-in extraction to OCaml source code is to extract towards an intermediate representation of the OCaml compiler. Such an approach has been verified [25].

Third, we need to trust that the BSML implementation on top of MPI — which contains OCaml code and C code — satisfies its specification as provided by BSML axioms in Coq. As mentioned in introduction, there is a verified sequential implementation that satisfies these axioms. Therefore, we can at least test the parallel implementation against the verified sequential implementation.

Fourth, we need to trust the OCaml compiler. A compiler is a large piece of software and always contain bugs unless it is formally verified. One possible way to remove the compiler from the trusted base would be to use the verified CakeML compiler [78]. However, it would require the implementation and verification of an extraction mechanism towards CakeML and a new implementation of BSML in CakeML.

4 SyDPaCC behind the Scenes

4.1 Reasoning about Functional Bulk Synchronous Parallel Programs

Bulk Synchronous Parallelism [82] is a structured model of parallelism. BSP programs run on BSP computers which are distributed memory machines with the ability to deliver point-to-point communications and possess a global synchronization unit. Any general purpose parallel machine (for e.g. a shared memory machine or a Cluster of PCs) can be seen as a BSP computer.

A BSP program is a sequence of *super-steps*. A super-step consists of three logically different phases. A visualization of a BSP program is given in Figure 5. In the computation phase, processors compute in parallel only using data they hold locally. In the data exchange phase, processors send/receive messages to/from other processors. Finally, the super-step ends with a global synchronization.

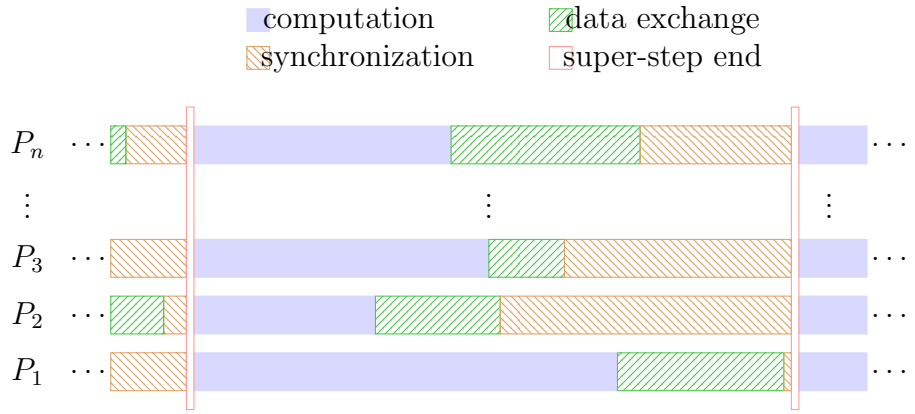

Figure 5: BSP *super-steps*

BSML is a pure functional programming library for OCaml [47] that supports the BSP model. It offers an abstract data type of distributed vectors and four operations to manipulate them. Addi-

$$
\begin{array}{l}
\texttt{mkpar} : (\texttt{int} \to \alpha) \to \alpha\,\texttt{par} \\
\texttt{mkpar}\ f = \langle f\,0,\ \ldots,\ f\,(p-1)\rangle \\
\texttt{apply} : (\alpha \to \beta)\texttt{par} \to \alpha\,\texttt{par} \to \beta\,\texttt{par} \\
\texttt{apply}\ \langle f_0,\ \ldots,\ f_{p-1}\rangle\ \langle v_0,\ \ldots,\ v_{p-1}\rangle = \langle f_0\ v_0,\ \ldots,\ f_{p-1}\ v_{p-1}\rangle \\
\texttt{proj} : \alpha\,\texttt{par} \to (\texttt{int} \to \alpha) \\
\texttt{proj}\ \langle v_0,\ \ldots,\ v_{p-1}\rangle = \texttt{function}\,0 \to v_0 \mid \ldots \mid p-1 \to v_{p-1} \\
\texttt{put} : (\texttt{int} \to \alpha)\texttt{par} \to (\texttt{int} \to \alpha)\texttt{par} \\
\texttt{put}\ \langle tosend_0,\ \ldots,\ tosend_{p-1}\rangle = \langle rcvd_0,\ \ldots,\ rcvd_{p-1}\rangle \\
\text{where } \forall\, src,\ dst.\ 0 \leq src, dst < p \Rightarrow rcvd_{dst}\ src = tosend_{src}\ dst
\end{array}
$$

Figure 6: BSML primitives

```
Definition replicate (A: Type)(x: A): par A :=
  mkpar(fun _ ⇒ x).

Definition parfun (A B: Type)(f: A→B)(v: par A): par B :=
  apply (replicate f) v.

Definition par_map (A B: Type)(f: A→B) : par(list A)→par(list B) :=
  parfun (map' f).
```

Figure 7: Parallel Map in Coq

tionally, it provides four constants that describe the performances of the BSP computer the BSML program runs on. The BSP cost model is out of the scope of this paper, we refer the interested reader to [6, 56]. Therefore, we use only one of these constants: `bsp_p` the number of processors of the BSP computer.

In the remaining of this section, we consider that the value of `bsp_p` is p a strictly positive integer. Let $\langle v_0,\ \ldots,\ v_{p-1}\rangle$ denotes a parallel vector. The only possible size for a parallel vector is p, hence there is one value per processor. The semantics of the four BSML primitives `mkpar`, `apply`, `proj` and `put` are given informally in Figure 6. `mkpar` builds a parallel vector from a function. `apply` applies a vector of functions (i.e. where at each processor the content of the vector is a function) to a vector of values. `proj` can be seen as the dual of `mkpar`. However, the result is a function defined only 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 processor i to processor j. `put` transposes this matrix and in the result, the cell j, i contains the message received by j from i. In BSML, these matrices are actually implemented as parallel vectors of functions.

The semantics of BSML in Coq is provided as a *module type*, i.e. it is an axiomatization of the informal semantics of Figure 6. To check that the axioms are non-contradictory, a sequential realization is provided by SyDPaCC. This module type allows to write and reason about BSML programs in Coq. For example, a parallel map working on distributed lists is given in Figure 7 where `map'` is a tail recursive version of sequential `List.map`.

SyDPaCC offers a set of such parallel functions, also called algorithmic skeletons and transformation theorems to automatically parallelize some classes of functions as compositions of algorithmic skeletons. Of course, the obtained parallel function is equivalent, in a sense given in the next subsection, to the initial sequential function.

4.2 Program Transformations

The support for program transformation and automatic parallelization relies on typeclasses:

- a typeclass for type correspondence (`TypCorr`): a type `A` is said to correspond to a type `B` if and only if there exists a surjective operation `join: B→A`. Intuitively it means that for any value `a` of `A`, there exists at least a value `b` of `B` corresponding to `a`. An example of type correspondence is the correspondence between a sequential list and a distributed list. A distributed list can be seen as p lists distributed on p processors. The `join` operation can be the concatenation of these lists (if the distribution is circular then the `join` operation should be something else). `List.app` is indeed surjective: a list `l` of size n can be divided into p lists of size about n/p in such a way that their concatenation is `l`. In this case the distribution is balanced. But of course it is not the only distribution. Uneven distributions are possible, for example the initial list on processor 0 and the empty list on other processors. In general the type `B` to which `A` corresponds has several ways to represent a value of type `A`.

- a typeclass for function correspondence (`FunCorr`): if type `A` corresponds to type `A'` (with `join_A`) and type `B` correspond to type `B'` (with `join_B`), then a function `f: A→B` corresponds to a function `f': A'→B'` if and only if `∀ (a': A'), join_B (f' a') = f (join_A a')`.

These notions of correspondences are composable: if a type `A` corresponds to a type `A'` and `A'` corresponds to `A''` then `A` corresponds to `A''`. This extends to function `f`, `f'` and `f''`. These compositions are called "vertical" compositions. If a function `f` corresponds to a function `f'` and `g` correspond to `g'` then `f ∘ g` corresponds to `f' ∘ g'` where $\circ$ is usual function composition. This kind of composition is called "horizontal" composition.

All these compositions are expressed as instances of the classes `TypCorr` and `FunCorr`. These instances can be thought as rules for the instance resolution mechanism.

Then there are instances that can be thought as facts for the Prolog-like instance resolution mechanism. For example, sequential lists correspond to distributed lists, sequential map on lists corresponds to the map algorithmic skeleton on distributed lists, written as (proof omitted):

```
Instance  map_par_map (A B: Type)(f: A→B):
  FunCorr (map f) (par_map f).
```

and sequential reduce corresponds to the reduce algorithmic skeleton on lists.

The `parallel` function used in Section 3 is defined as:

```
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.
```

The ` before an argument means that if the type of the argument uses names that are not defined yet, these names are automatically added as implicit arguments. For example, arguments `(A: Type)` `(B: Type)` are added automatically because the type of `f` is `A→B`. Using `{}` instead of `()` for an argument means that this argument is implicit and Coq will try to infer it. That is why even though the definition of `parallel` looks like this function has four arguments (`f`, `ACorr`, `BCorr` and `fCorr`)[1], we applied it to only one argument in Section 3. Without these implicit arguments, this function would not be very interesting because it simply returns one of its arguments: `fp`. But because `fCorr` is an implicit argument and the type of this argument is a typeclass, Coq tries to build an instance of this typeclass automatically, meaning it tries to build a parallel function `fp` that corresponds semantically to sequential function `f`.

[1] and actually, because of the use of ` it has many more, all implicit

To do so, it uses parametric instances such as the composition of instances and the first homomorphism theorem, and non-parametric instances such as the correspondences of sequential and parallel map and reduce. Therefore, a sequential homomorphic function can be automatically parallelized as a composition of the algorithmic skeletons map and reduce: that is how the `count` example is automatically parallelized.

5 A Tour of SyDPaCC Transformations and Applications

The early work on SyDPaCC started with contributions to the definition and formalization in Coq of BSP homomorphisms [28] which was not initially executable at scale. It was improved and made executable. A quite complicated and preliminary form of transformation based on typeclasses was proposed [29, 79], in a framework named SDPP. A parallel version of the tower building problem [29] as well as the all nearest smaller values [57] were developed using BSP homomorphisms and the associated transformation theorems.

While working on a class for a summer school [58], we realized that the transformation infrastructure of SDPP could be hugely simplified and that list homomorphisms and the homomorphisms theorems [30] fitted very well the transformation approach. We therefore started from scratch a new framework coined SyDPaCC which offered initially list homomorphism-based transformations [59]. SyDPaCC offers a few applications using these transformations including the parallelization of the maximum prefix sum problem.

We then extended SyDPaCC to deal with functions that are not necessarily list homomorphisms but a more general class [50]. The functions in this class are transformed using the so-called diffusion theorem [38] and the associated accumulate algorithmic skeleton [41]. The application is a parallel version of the bracket matching problem.

More recently, we ported and improved (removing all the non-tail recursive functions) BSP homomorphisms (BH) into SyDPaCC [54]. We parallelized a heat diffusion simulation using BH transformations.

The version of SyDPaCC described in this paper has around 7kLoC of Coq code including 4kLoC of proof scripts. The parallel implementation of BSML primitives is about 300 LoC of OCaml and 240 LoC of C. The considered applications so far are still modest. Code extraction yields code that is more verbose than hand-written code. The largest examples are in the range of 100-200 LoC of hand-written code including the algorithmic skeletons and the application.

One set of SDPP transformations has not yet been included into SyDPaCC (but will be): the so-called GTA approach [20]. In this approach, a user defines an inefficient specification of her problem as a composition of a generator of all the candidate solutions, a tester of valid solutions, and an aggregator to combine the solutions. Two transformation theorems (filter embedding and semiring fusion [19]) allow obtaining efficient parallel programs as compositions of map and reduce. The approach is applied to the knapsack problem.

6 Related Work

Bird introduced constructive algorithmics [5] and the literature on the matter is rich and included parallel programming (for e.g. [39, 32, 16, 62, 12]). Most of the work is done on paper while more recently interactive theorem proving is used (for e.g. [63]) but not in the parallel case.

There are contributions that formalize some aspects of parallel programming, but to our knowledge these approaches do not allow to directly obtain executable code as we do with SyDPaCC: Swierstra [77] formalized mutable arrays with explicit distributions in Agda ; BSP-Why [27] allows to deductively verify imperative BSP programs, but these are models of C BSPlib [36] programs

rather than executable code ; the Data Parallel C programming language was formalized with Isabelle/HOL [14] and the tool generated Isabelle/HOL expressions representing the parallel of the program; Grégoire and Chlipala provide a small parallel language and its semantics and proves correct optimizations of stencil-based computations [34].

Studying, from the functional programming perspective, frameworks such as Hadoop MapReduce [44, 60] and Apache Spark [1, 9] is related to SyDPaCC as we may take a similar approach to be able to extract MapReduce or Spark programs from Coq. Ono et al. [64] used Coq to verify MapReduce programs and either extract Haskell code fed to Hadoop Streaming or directly write Java programs annotated with JML and used Krakatoa [24] to generate Coq lemmas. This work is less systematic and automated than SyDPaCC. Also related to Apache Spark, Philippe et al. [68] presented a formalized model transformation engine in Coq, but there is neither automated program transformation nor automatic code extraction.

SyDPaCC supports deterministic parallel programming which is also the focus of [40, 7, 71]. The context is however very different. These contributions consider sequential imperative programs with contracts and annotations and transform both the programs and the annotations so that if the sequential program respects its contract, so will the parallel program. The approaches consider more low-level programs and annotations are necessary which makes SyDPaCC more high-level, but the obtained parallel imperative code is much more efficient than SyDPaCC code. These approaches also focus on OpenMP compiler directives or GPU code which limits the applicability to shared memory machines.

7 Conclusion and Future Work

SyDPaCC currently handles the automatic parallelization of list homomorphisms, of functions performing accumulative computations, of BSP homomorphisms, as well as their compositions. The obtained code is functional bulk synchronous parallel code that we have demonstrated to be scalable by running it on parallel machines

There are several directions of future work for SyDPaCC:

- Work on new transformations to handle the automatic parallelization of new classes of functions, including functions that manipulate other data-structures than lists, such as trees or graphs for which there is preliminary work [66, 67].

- Work on other back-ends than OCaml and BSML, in particular Scala and Spark for which we have a not yet fully automated solution [53]. An automated solution requires a Scala code extractor from Coq code more general than the one we used [17]. To design and implement such a code extraction plugin, we plan to rely on MetaCoq [75].

  We could also consider extraction towards Python and algorithmic skeletons libraries such as PySke [52], and towards C++ and algorithmic skeleton libraries, such as SkePU [22], Muesli [35], SkeTo [18] or OSL [46, 45]. To be very efficient we may to be consider code extraction that do use mutation and explicit memory allocation and deallocation.

- The transformation mechanism relies on typeclass resolution in Coq. It returns the first solution it finds, not necessarily the best one in terms of parallel performances. We plan to explore an extraction mechanism that takes into account the parallel performances. We would require to leverage the BSP performance model, reason about performances in Coq (there exists work on complexity reasoning in Coq for sequential programs, for e.g. [83]) and to design and implement a search that takes into account performances. The implementation could use ELPI an implementation of λProlog extended with Constraint Handling Rules that can be used to manipulate Coq terms [33].

Acknowledgements. This work was supported by the project AcceptAlgo funded by Region Centre Val-de-Loire.

References

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

[2] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. Opening polyhedral compiler's black box. In *Code Generation and Optimization (CGO)*, page 128–138, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450337786. doi:10.1145/2854038.2854048.

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

[4] Richard Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans Program Lang Syst*, 6(4):487–504, October 1984. doi:10.1145/1780.1781.

[5] Richard Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[6] R. Bisseling. *Parallel Scientific Computation. A Structured Approach using BSP and MPI*. Oxford University Press, 2004.

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

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Programming Language Design and Implementation (PLDI)*, pages 101–113, New York, USA, 2008. ACM. doi:10.1145/1375581.1375595.

[9] Yu-Fang Chen, Chih-Duo Hong, Ondrej Lengál, Shin-Cheng Mu, Nishant Sinha, and Bow-Yaw Wang. An executable sequential specification for Spark aggregation. In *Networked Systems (NETSYS)*, volume 10299 of *LNCS*, pages 421–438, 2017. doi:10.1007/978-3-319-59647-1_31.

[10] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2014.

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

[12] Murray Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. In Gerhard R. Joubert, Denis Trystram, Frans J. Peters, and David J. Evans, editors, *Parallel Computing: Trends and Applications, PARCO 1993*, pages 489–492. Elsevier, 1994.

[13] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles & Practice of Parallel Programming (PPOPP)*, pages 1–12, New York, NY, USA, 1993. ACM. doi:10.1145/155332.155333.

[14] M. Daum. Reasoning on Data-Parallel Programs in Isabelle/Hol. In *C/C++ Verification Workshop*, 2007. doi:http://www.cse.unsw.edu.au/ rhuuck/CV07/program.html.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

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

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

[18] K. Emoto and K. Matsuzaki. An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo. *Int J Parallel Prog*, 2013. doi:10.1007/s10766-013-0263-8.

[19] Kento Emoto, Sebastian Fischer, and Zhenjiang Hu. Generate, Test, and Aggregate – A Calculation-based Framework for Systematic Parallel Programming with MapReduce. In *ESOP*, volume 7211 of *LNCS*, pages 254–273. Springer, 2012. doi:10.1007/978-3-642-28869-2_13.

[20] Kento Emoto, Frédéric Loulergue, and Julien Tesson. A Verified Generate-Test-Aggregate Coq Library for Parallel Programs Extraction. In *Interactive Theorem Proving (ITP)*, number 8558 in LNCS, pages 258–274, Wien, Austria, 2014. Springer. doi:10.1007/978-3-319-08970-6_17.

[21] J. Enmyren and C. Kessler. SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. In *4th workshop on High-Level Parallel Programming and Applications (HLPP)*. ACM, 2010.

[22] August Ernstsson, Dalvan Griebler, and Christoph W. Kessler. Assessing application efficiency and performance portability in single-source programming for heterogeneous parallel systems. *Int. J. Parallel Program.*, 51(1):61–82, 2023. doi:10.1007/S10766-022-00746-1.

[23] Paul Feautrier. Some efficient solutions to the affine scheduling problem: One-dimensional time. *Int J Parallel Prog*, 21(5):313–347, 1992. doi:10.1007/BF01407835.

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

[25] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. Verified extraction from coq to ocaml. *Proc. ACM Program. Lang.*, 8(PLDI), jun 2024. doi:10.1145/3656379.

[26] Frédéric Gava. Une bibliothèque certifiée de programmes fonctionnels BSP. *Technique et Science Informatiques*, 25(10):1261–1280, 2006.

[27] Frédéric Gava, Jean Fortin, and Michaël Guedj. Deductive Verification of State-Space Algorithms. In *IFM*, volume 7940 of *LNCS*, pages 124–138. Springer, 2013. doi:10.1007/978-3-642-38613-8_9.

[28] Louis Gesbert. *Développement systématique et sûreté d'exécution en programmation parallèle structurée*. PhD thesis, University Paris Est, LACL, 2009. URL `http://tel.archives-ouvertes.fr/tel-00481376`.

[29] Louis Gesbert, Zhenjiang Hu, Frédéric Loulergue, Kiminori Matsuzaki, and Julien Tesson. Systematic Development of Correct Bulk Synchronous Parallel Programs. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 334–340. IEEE, 2010. doi:10.1109/PDCAT.2010.86.

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

[31] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software, Practrice & Experience*, 40(12):1135–1160, 2010. doi:10.1002/spe.1026.

[32] Sergei Gorlatch and Holger Bischof. Formal Derivation of Divide-and-Conquer Programs: A Case Study in the Multidimensional FFT's. In D. Mery, editor, *Formal Methods for Parallel Programming: Theory and Applications*, pages 80–94, 1997.

[33] Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality tests, automatically: Deriving eqType instances for Jasmin's data types with Coq-Elpi. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 167–181. ACM, 2023. doi:10.1145/3573105.3575683.

[34] Thomas Grégoire and Adam Chlipala. Mostly automated formal verification of loop dependencies with applications to distributed stencil algorithms. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving (ITP)*, volume 9807 of *LNCS*, pages 167–183. Springer, 2016. doi:10.1007/978-3-319-43144-4_11.

[35] Nina Herrmann and Herbert Kuchen. Distributed calculations with algorithmic skeletons for heterogeneous computing environments. *Int. J. Parallel Program.*, 51(2-3):172–185, 2023. doi:10.1007/S10766-022-00742-5.

[36] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.

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

[38] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating Efficient Parallel Programs. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, pages 85–94. ACM, January 22-23 1999.

[39] Zhenjiang Hu, Hidewaki Iwasaki, and Masato Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Trans Program Lang Syst*, 19(3):444–461, 1997. ISSN 0164-0925. doi:10.1145/256167.256201.

[40] Marieke Huisman, Stefan Blom, Saeed Darabi, and Mohsen Safari. 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*, pages 365–80, Berlin, Heidelberg, 2018. Springer-Verlag. ISBN 978-3-030-03417-7. doi:10.1007/978-3-030-03418-4_22.

[41] Hideya Iwasaki and Zhenjiang Hu. A new parallel skeleton for general accumulative computations. *Int. J. Parallel Program.*, 32(5):389–414, 2004. doi:10.1023/B:IJPP.0000038069.80050.74.

[42] J. Jájá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[43] Ulrike Klusik, Rita Loogen, Steffen Priebe, and Fernando Rubio. Implementation Skeletons in Eden: Low-Effort Parallel Programming. In Markus Mohnen and Pieter W. M. Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers*, LNCS 2011, pages 71–88. Springer, 2000.

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

[45] Joeffrey Légaux, Zhenjiang Hu, Frédéric Loulergue, Kiminori Matsuzaki, and Julien Tesson. Programming with BSP Homomorphisms. In *Euro-Par Parallel Processing*, number 8097 in LNCS, pages 446–457, Aachen, Germany, 2013. Springer. doi:10.1007/978-3-642-40047-6_46.

[46] Joeffrey Légaux, Sylvain Jubertie, and Frédéric Loulergue. Development Effort and Performance Trade-off in High-Level Parallel Programming. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 162–169, Bologna, Italy, 2014. IEEE. doi:10.1109/HPCSim.2014.6903682.

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

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

[49] R. Loogen, Y. Ortega-Mallen, and R. Pena-Mari. Parallel Functional Programming in Eden. *J Funct Program*, 3(15):431–475, 2005. doi:10.1017/S0956796805005526.

[50] Frédéric Loulergue. A verified accumulate algorithmic skeleton. In *Fifth International Symposium on Computing and Networking (CANDAR)*, pages 420–426, Aomori, Japan, November 19-22 2017. IEEE. doi:10.1109/CANDAR.2017.108.

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

[52] Frédéric Loulergue and Jolan Philippe. Automatic Optimization of Python Skeletal Parallel Programs. In *Algorithms and Architectures for Parallel Processing (ICA3PP)*, LNCS, pages 183–197, Melbourne, Australia, 2019. Springer. doi:10.1007/978-3-030-38991-8_13.

[53] Frédéric Loulergue and Jolan Philippe. Towards verified scalable parallel computing with Coq and Spark. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP)*, pages 11–17, New York, NY, USA, 2023. ACM. doi:10.1145/3605156.3606450.

[54] Frédéric Loulergue and Julien Tesson. Verified Parallel Programming in Coq with Bulk Synchronous Parallel Homomorphisms. In *17th International Symposium on High-Level Parallel Programming and Applications (HLPP)*, Pisa, Italy, July 2024. hal: hal-04597523.

[55] Frédéric Loulergue, Gaétan Hains, and Christian Foisy. A Calculus of Functional BSP Programs. *Sci Comput Program*, 37(1-3):253–277, 2000. doi:10.1016/S0167-6423(99)00029-5.

[56] Frédéric Loulergue, Frédéric Gava, and David Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In *International Conference on Computational Science (ICCS)*, volume 3515 of *LNCS*, pages 1046–1054. Springer, 2005. doi:10.1007/11428848_-132.

[57] Frédéric Loulergue, Simon Robillard, Julien Tesson, Joeffrey Légaux, and Zhenjiang Hu. Formal Derivation and Extraction of a Parallel Program for the All Nearest Smaller Values Problem. In *ACM Symposium on Applied Computing (SAC)*, pages 1577–1584, Gyeongju, Korea, 2014. ACM. doi:10.1145/2554850.2554912.

[58] Frédéric Loulergue, Wadoud Bousdira, and Julien Tesson. Calcul de programmes parallèles avec Coq. In Nicolas Ollinger, editor, *Informatique Mathématique une photographie en 2015*, collection Alpha, pages 87–134. CNRS Éditions, 2015.

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

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

[61] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*, November 2023. URL `https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf`.

[62] A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In Zhong Shao and Benjamin C. Pierce, editors, *POPL'09*, pages 177–185. ACM, 2009. doi:10.1145/1480881.1480905.

[63] Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming in Agda: Dependent types for relational program derivation. *J Funct Program*, 19(5):545–579, 2009. doi:10.1017/S0956796809007345.

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

[65] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

[66] Jolan Philippe and Frédéric Loulergue. Parallel programming with Coq: Map and reduce skeletons on trees. In *ACM Symposium on Applied Computing (SAC)*, pages 1578–1581. ACM, 2019. doi:10.1145/3297280.3299742.

[67] Jolan Philippe, Frédéric Loulergue, and Wadoud Bousdira. Formalization of a Big Graph API in Coq (Poster). In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 893–894, Genoa, Italy, 2017. IEEE. doi:10.1109/HPCS.2017.140.

[68] Jolan Philippe, Massimo Tisi, Hélène Coullon, and Gerson Sunyé. Executing certified model transformations on Apache Spark. In *14th International Conference on Software Language Engineering (SLE)*, pages 36–48, New York, NY, USA, 2021. ACM. doi:10.1145/3486608.3486901.

[69] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative Optimization in the Polyhedral Model: part II, Multidimensional Time. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 90–100. ACM, 2008. doi:10.1145/1375581.1375594.

[70] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003. doi:10.1007/978-1-4471-0097-3.

[71] Ömer Şakar, Mohsen Safari, Marieke Huisman, and Anton Wijs. Alpinist: An annotation-aware gpu program optimizer. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 332–352, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-030-99527-0_18.

[72] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.

[73] M. Sozeau and N. Oury. First-Class typeclasses. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume LNCS 5170, pages 278–293. Springer, 2008.

[74] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi:10.1145/3371076.

[75] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The metacoq project. *J. Autom. Reason.*, 64(5):947–999, 2020. doi:10.1007/S10817-019-09540-0.

[76] Bas Spitters and Eelis Van der Weegen. Typeclasses for mathematics in type theory. *Mathematical Structures in Computer Science*, 21:795–825, 7 2011. doi:10.1017/S0960129511000119.

[77] Wouter Swierstra. More dependent types for distributed arrays. *Higher-Order and Symbolic Computation*, 23(4):489–506, 2010. doi:10.1007/s10990-011-9075-y.

[78] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *International Conference on Functional Programming (ICFP)*, page 60–73, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342193. doi:10.1145/2951913.2951924.

[79] Julien Tesson. *Environnement pour le développement et la preuve de correction systématiques de programmes parallèles fonctionnels*. PhD thesis, LIFO, University of Orléans, November 2011. URL `http://hal.archives-ouvertes.fr/tel-00660554/en/`.

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

[81] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`.

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

[83] E. van der Weegen and J. McKinna. A Machine-checked Proof of the Average-case Complexity of Quicksort in Coq. In Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors, *Types for Proofs and Programs, International Conference (TYPES 2008)*, LNCS 5497, pages 256–271. Springer, 2008. doi:10.1007/978-3-642-02444-3.