Automatic Optimization of Python Skeletal Parallel Programs

Frédéric Loulergue, Jolan Philippe

Automatic Optimization of Python Skeletal Parallel Programs

Frédéric Loulergue[1][0000−0001−9301−7829] and Jolan Philippe[1,2][0000−0001−8759−4566]

[1] School of Informatics Computing and Cyber Systems
Northern Arizona University, Flagstaff, AZ, USA
frederic.loulergue@nau.edu
[2] IMT Atlantique, Inria, LS2N, UMR CNRS 6004
F-44307 Nantes, France
jolan.philippe@imt-atlantique.fr

Abstract. Skeletal parallelism is a model of parallelism where parallel constructs are provided to the programmer as usual patterns of parallel algorithms. High-level skeleton libraries often offer a global view of programs instead of the common Single Program Multiple Data view in parallel programming. A program is written as a sequential program but operates on parallel data structures. Most of the time, skeletons on a parallel data structure have counterparts on a sequential data structure. For example, the map function that applies a given function to all the elements of a sequential collection (e.g., a list) has a map skeleton counterpart that applies a sequential function to all the elements of a distributed collection. Two of the challenges a programmer faces when using a skeleton library that provides a wide variety of skeletons are: which are the skeletons to use, and how to compose them? These design decisions may have a large impact on the performance of the parallel programs. However, skeletons, especially when they do not mutate the data structure they operate on, but are rather implemented as pure functions, possess algebraic properties that allow to transform compositions of skeletons into more efficient compositions of skeletons. In this paper, we present such an automatic transformation framework for the Python skeleton library PySke and evaluate it on several example applications.

Keywords: Algorithmic skeletons · Program transformation · Python.

1 Introduction

Context and Motivation Most computing devices are now parallel architectures, and more and more data is produced and analyzed. However, writing programs for parallel architectures remains difficult compared to writing sequential programs. Parallel programming may increase programs performances but also hinders programming productivity.

The skeletal approach proposed by Murray Cole [13] consists in defining a program using computational patterns that have parallel implementations. In other words, programmers do not have to think about parallel aspects of their program, but only about how to write their program using already implemented patterns.

Skeletons are often implemented as higher-order functions, providing both flexibility and expressivity for the developers. Generally, a skeleton on a distributed data-structure is associated with a sequential function for a corresponding sequential data-structure. Such a high-level approach makes parallelism more affordable, even for non-expert parallel developers. However, most of the libraries handling parallel data-structures have been developed in order to be used by advanced users: the use of low-level languages or low-level primitives for parallelism, *etc*. For example, SkeTo [15], SkePu [16], OSL [18], or Muesli [11] are skeleton libraries written in C++. Their main advantage is performance. However, C++ itself is a complex language compared to Python and its dynamic typing and high-level features.

PySke [20], whose name is composed by *Python* and *Skeleton*, aims at providing computational patterns for lists and trees in both sequential and parallel, making the parallel aspects of programs as abstract as possible. PySke targets users without a deep background in computer science. There are two main challenges for such developers when using a skeleton library that provides a wide variety of skeletons: the choice of skeletons to compose the application, and the way these skeletons are composed. These application design choices may have a big impact on the performance of the application. Nonetheless, compositions of skeletons, especially on immutable data-structures, obey some algebraic laws that can be used for optimizing skeleton compositions.

Contributions In [21] we evaluated such optimizations. However, the transformations were not automated but hand-written. In this paper, we propose a fully automatic optimizer of PySke programs based on program transformation, and we evaluate it on some example applications.

Outline The paper is organized as follows. In Section 2, we discuss work related to our contribution. We give an overview of PySke in Section 3. Section 4 is devoted to the design of the new program transformation feature of PySke. We evaluate the performance of optimized example programs in Section 5. We conclude and discuss future work in Section 6.

2 Related Work

The transformation rules we use to optimize programs come from the Bird-Merteens Formalism (BMF) tradition [6,7]. For example, the map function/skeleton – that applies the same function to all the elements of a list – satisfies the following property: The composition of two map can be transformed into a single use of map. Considering two functions f and g, $(map\ f) \circ (map\ g)$ can be transformed into $map\ (f \circ g)$, where $\circ$ is function composition. This transformation removes the allocation and transversal of an intermediate list, thus optimizes the initial program.

To our knowledge, PySke is the only algorithmic skeleton library for Python. But there are many skeleton libraries or eDSL for other programming languages as well as programming languages specifically designed around the concept of algorithmic skeletons. In this section, we discuss only libraries and languages where program transformation is used as a means to optimize parallel programs.

For skeletons libraries, there are two main categories:

- Libraries where the optimization is done at compile time using meta-programming techniques either provided directly by the host language, or extensions, or abusing some other language features;
- Libraries where the optimization is done at execution time, either by encoding the skeleton expressions in a data structure similar to an abstract syntax tree and perform the optimization on this structure, or by using the introspection/reflection features of the host language.

In the first category, the Delite [22] framework for Scala can be considered as a skeletal parallelism approach. It features a set of data structures and mostly classical skeletons on them: map and variants, reduce and variants, filter, sort; and one less usual skeleton: group-by, as Delite has dictionaries as one of its supported data structures. Delite provides compile-time optimization through staged programming. Delite targets heterogeneous architectures CPU/GPU but only shared memory architectures. Several C++ libraries [15, 17, 18] are also in this category: they abuse the template feature of C++ to provide compile-time optimization. Note that in these libraries, the transformations are quite ad-hoc, and the main goal is to remove intermediate data-structures and loop traversals. The map transformation rule given above is actually such a transformation. But more complex or high-level transformations are usually not handled by these libraries that often only optimize the sequential parts of the programs but stop their optimizations as soon as there are communications. To our knowledge, there is no compile-time staging framework for Python thus such an approach cannot be used for PySke without writing a specific compiler (possibly a source-to-source compiler).

In the second category, Accelerate [10,12] – a Haskell framework for programming GPUs – features classical data-parallel skeletons (map and variants, reduce, scan and permutation skeletons) on multi-dimensional arrays and streams. Optimization of the GPUs programs are done at runtime. PySke also belongs to this category: we optimize compositions of skeletons at runtime, and we rely on a data-structure representing these compositions to perform the transformations. In this category, another approach is to rely also on the reflection features of the host language. Lithium [1] is a structured parallelism framework that leverages Java reflection to optimize programs. Introspection and reflection also exist in Python, and we do rely on such features in the extension of PySke considered in this paper.

Finally, there are skeleton languages. The main advantage of such an approach is that the way optimizations are performed is not limited to what the host language allows. The main drawbacks are that specific compilers should be designed and implemented, and new languages do not have as large libraries as mainstream languages do, which hinders their adoption. An intermediate approach is to have a specific language for writing the composition of skeletons and use a usual sequential language for writing the arguments to these skeletons. This is less flexible than having a completely new language but mitigates the drawbacks mentioned. P3L [19] associated with the FAN transformation framework [2, 4] is a representative of this approach. While being very efficient, this kind of approach is much more complex to deploy than a Python library and requires a significant effort from the programmer.

3 An Overview of PySke

PySke is a library for Python currently implemented on top of MPI and *mpi4py* [14]. Note that the programming model of PySke is independent of the underlying communication library.

PySke offers a *global view* of programs. A PySke program is written (and read) as a sequential program but it operates on parallel data structures. This aspect is very different from the SPMD paradigm of MPI where most of the time a program is actually parametrized by the process identifier (returned by the method `Get_rank` in mpi4py), and the global parallel program should be understood as the parallel composition of instantiations – for all possible process identifiers – of this parametrized sequential program. This "par of seq" structure is more complicated to deal with than the "seq of par" structure that global view offers [8].

For readers familiar with MPI, PySke can be thought as a library of collectives. There is a major difference: arguments to MPI collectives have regular C types but the collection on all processors of the values of these sequential types may be thought as a parallel data structure whereas in PySke there are classes dedicated to parallel data structures. In PySke, the type of a value indicated whereas this value is sequential or parallel. In MPI there is no parallel type, and it may be difficult (and it is an undecidable problem in general) to know if a value is sequential or should be thought as being part of a distributed data structure.

```
n   = data.length()
avg = data.reduce(add) / n
def f(x): return (x-avg) ** 2
var = data.map(f).reduce(add) / n
```

Fig. 1: Variance in PySke

When possible, PySke offers the same methods for both a sequential data structure and the corresponding parallel data structure. The code in Figure 1 computes the variance of a discrete random variable `data`. This code is valid when `data` is either an instance of `SList`, i.e. a sequential list, or an instance of `PList`, i.e. a parallel list distributed on all the processors of the parallel machine running the program.

This example shows two classical skeletons: `map` that applies the same function (here `f`) to all the elements of a data-structure, and `reduce` that uses a binary *associative* operation (here `add` as defined in the Python module `operator`) to "sum" all the elements of a list using the binary operation. For their implementations in `PList`, `map` does not require any communication to be executed and `reduce` does require some communication: first, the partial sums are computed on each processor, then these partial sums are sent to all processors (total exchanged), and finally, the final sum is computed.

The user of PySke can see a `PList` as a usual list (`SList` extends the Python lists with additional methods). The implementation of `PList` is however an MPI one and therefore follows the SPMD approach. On each processor, a `PList` is actually composed of several fields: the content (a sequential list specific to each processor), the global size (same value on all processors), the distribution (the same list on each processor, this list contains the lengths of all local contents), and the index of the first element of the local list in the global list (for example at processor 2, this value would be 9 if processor 0 has a local list of 5 elements, and processor 1 a local list of 4 elements). All the methods provided by `PList` ensure that the content of these fields stay what we have just described: This may require communications for some skeletons which intuitively do not need them to perform the intended computations on the content of the parallel list.

For example, the `filter` skeleton – that takes as argument a predicate `p` and returns a parallel list where only the values satisfying the predicate `p` are kept – does not require any communication to compute the content of the returned parallel list. It is enough to perform a sequential `filter` on each of the local contents. However, if the skeleton only performed this local filtering, the global size, the local index, and the distribution would no longer represent the actual global size, local index, and distribution.

The current set of list skeletons contains [20]: `reduce`, `map` and variants (`mapi`, `map2`, `zip`), `scan` and variants, `get_partition`, `flatten`, and `balance`.

For example, `SList([1,2,3]).scan(add)` is `[0, 1, 3, 6]`.

`get_partition` makes the way the data-structure is distributed (or partitioned) visible in the value itself. For example if the global view of a parallel list `pl` is `[1, 2, 3, 4, 5, 6]` and the distribution on 4 processors is `[2, 2, 1, 1]`, then the global view of `pl.get_partition()` is `[[1, 2], [3, 4], [5], [6]]`. `flatten` is the inverse operation (and requires communications).

These two skeletons can be used to implement a `filter` skeleton:

```python
def filter(self, p):
  return self.get_partition().
         map(lambda l: l.filter(p)).
         flatten()
```

After a call to `filter` the resulting parallel list may be unbalanced, i.e., some processors may contain much more elements than some others. The `balance` skeleton redistributes values such that the parallel list is evenly distributed, i.e., any processor has at most one more element than each other processor.

PySke also features a set of skeletons on trees. There are three tree data-structures: binary trees, linearized trees, parallel trees. The two first structures are sequential structures, while the last one is a parallel data structure. All of them represent binary trees.

Figure 2 is the code necessary to count the number of elements of a parallel tree that satisfy a predicate `p`. The code differs for a value of class `BTree` of sequential binary trees (variable `bt`) and a value of class `PTree` of parallel trees (variable `pt`). This is due to the fact that in order to be executed in parallel the reduction of a tree using an operator $\oplus$, this operator should satisfy a property called the *closure* property that allows expressing the operator using 4 auxiliary functions. This property is in a way what corresponds to associativity for lists. `map` on trees needs two arguments: a function to apply to the leaf values and a function to apply to the values at the nodes. There

```
def id(x): return x
def sum(x, y, z): return x + y + y
def f(x): return 1 if p(x) else 0
count_bt = bt.map(f,f).reduce(sum)
count_pt = pt.map(f,f).reduce(sum, id, sum, sum, sum)
```

Fig. 2: Count on Trees in PySke

```
from pyske.core.list.plist import PList as PL
from pyske.core.opt.list import PList
# ...

pl1 = PL.init(rand, size)
pl2 = PL.init(rand, size)

def dot_product1(pl1, pl2):
    dot = pl2.zip(pl1).map(uncurry(mul)).reduce(add, 0)
    return dot

def dot_product2(pl1: PL, pl2: PL):
    pl1 = PList.wrap(pl1)
    pl2 = PList.wrap(pl2)
    return dot_product1(pl1, pl2).run()
```

Fig. 3: PySke with Automatic Optimization

are two functions on trees that correspond to `scan` on lists: downwards accumulation (`dacc`) and upwards accumulation (`uacc`). More details are provided in [20].

4 PySke with Automatic Optimization

In order to support automatic optimization of PySke applications, the user programming interface needs to slightly change with respect to what we presented in the previous section. We present these changes in Section 4.1. The remaining sub-sections are devoted to the design of the automatic optimization mechanism of PySke which is rule-based. It therefore relies on concepts from term rewriting systems [3]: we give a short overview in Section 4.2, and explain how we implemented support for such systems in a generic way in Python in Section 4.3. This generic framework is then instantiated to deal with PySke skeletons (Section 4.4).

4.1 User Programming Interface

We illustrate the differences between the previous API and the new one through the example of Figure 3: the computation of the dot product of two vectors represented as two parallel lists.

`dot_product1` is the version written using only the features presented in Section 3. It uses the data structure `PList` (imported as `PL` in this example) of the module `pyske.core.list.plist` and its skeletons.

`dot_product2` is the version written using the new API: the `PList` class of the module `pyske.core.opt.list`. First note that `dot_product2` can still take as input usual parallel lists: we just need to use a wrapper that transforms a `PL` into a `PList` (Lines 13–14). If `dot_product` were to take as argument parallel lists of type `PList` then no wrapping would be necessary. The only other difference with the previous API is that it is necessary to call the method `run` to launch the optimization of the skeleton composition and the execution of the optimized version. Note that to implement `dot_product2` we can reuse `dot_product1`: this makes the transition to the new version easier.

`uncurry` is a higher-order function that transforms a function taking as input two formal parameters into a function taking only one formal parameter, this parameter being a pair. `curry` does the inverse transformation. Compositions of such functions are optimized too.

The performances of this example are discussed in Section 5.

4.2 Term Rewriting

Basically, PySke programs are skeleton expressions. Therefore, they can be represented as terms, and we can use rules to transform these terms. Our optimization mechanism is thus a term rewriting system [3].

Terms A *signature* Σ is a set $\mathcal{F}$ of function symbols with a function $ar : \mathcal{F} \rightarrow \mathbb{N}$ that for each function symbol f gives its *arity*. We call symbol functions of arity 0 *constants*.

We assume a countable set $\mathcal{X}$ of variables such that $\mathcal{X} \cap \mathcal{F} = \emptyset$.

The set $\mathcal{T}$ of terms over $\mathcal{X}$ and Σ is the smallest set such that:

– for all variable $x \in \mathcal{X}$, $x \in \mathcal{T}$,
– for all function symbol $f \in \mathcal{F}$, number n such that $n = ar(f)$, and for all terms $t_1, \ \ldots, \ t_n$ then $f(t_1, \ldots, t_n) \in \mathcal{T}$.

The set of variables of a term t can be computed by the following function $\mathcal{V}$:

$$\begin{cases} \mathcal{V}(x) & = \{x\} \\ \mathcal{V}(f(t_1, \ldots, t_n)) & = \cup_{k=1}^{n} \mathcal{V}(t_k) \end{cases}$$

A *closed* term is a term t that does not contain any variable, i.e. $\mathcal{V}(t) = \emptyset$. A term is said *linear* if each $x \in \mathcal{V}(t)$ appears at most once in t.

A *substitution* σ is a partial function from $\mathcal{X}$ to $\mathcal{T}$. It can be extended to $\mathcal{T}$ as follows: for any $t \in \mathcal{T}$, either $t \in \mathcal{X}$ and we can directly use σ, or there exist a function symbol f of arity n, and n terms $t_1, \ \ldots, \ t_n$ such that $t = f(t_1, \ldots, t_n)$. We can then define: $\sigma(t) = f(\sigma(t_1), \ldots, \sigma(t_n))$.

$\sigma_\perp$ denotes the substitution undefined everywhere, and for a substitution σ, a variable x and a term t, the update $\sigma[x \mapsto t]$ is the substitution defined by:

$$\sigma[x \mapsto t](y) = \begin{cases} \sigma(y) & \text{if } y \neq x \\ t & \text{if } y = x \end{cases}$$

Rules A *rule* is a pair of terms, denoted by $l \to r$, such that $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. A rule is *linear* if both l and r are linear.

Assuming t is a closed term and p is a linear term, we define the function $match(t, p)$ as follows:

$$\begin{cases} match(t, x) & = \sigma_\bot[x \mapsto t] \\ match(f(t_1, \ldots, t_n),\ g(t'_1, \ldots, t'_m)) = & \\ \quad merge(match(t_1, t'_1), \ldots, match(t_n, t'_n)) & \text{if } f = g \text{ and } n = m \\ match(f(t_1, \ldots, t_n),\ g(t'_1, \ldots, t'_m)) = \bot & \text{otherwise} \end{cases}$$

The function $merge$ returns $\bot$ if one of it arguments is $\bot$, otherwise it merges the substitutions in arguments. As p is supposed to be linear, each substitution maps variables that are different from the variables mapped by each other substitution.

The application of a rule $l \to r$ at the root of a closed term t proceeds as follows:

1. we check if the left-hand side of the rule matches t, i.e. we compute $\sigma = match(t, l)$,
2. if $\sigma = \bot$ then it means l does not match t, so the rule cannot be applied,
3. if σ is a valid substitution, the rule can be applied and then we obtain the term $\sigma(r)$.

A rule can be applied to sub-terms of a term t. In this case, if the matching succeeds, it is the matched sub-term that is replaced by $\sigma(r)$.

Example Let us consider the following signature: $\mathcal{F} = \{\texttt{map}, \texttt{reduce}, \texttt{pl}, \texttt{incr}, +, \circ\}$ with $ar(\texttt{map}) = 2$, $ar(\texttt{reduce}) = 2$, $ar(\texttt{pl}) = 0$, $ar(\texttt{incr}) = 1$, $ar(+) = 2$, $ar(\circ) = 2$, and we use $+$ and $\circ$ as infix operations. They represent respectively the addition and function composition.

Let us consider the following term:

$$t = \texttt{reduce}(\texttt{map}(\texttt{map}(\texttt{pl}, \texttt{incr}), \texttt{incr}), +)$$

that can be thought as a representation of the following PySke program:

```
pl.map(incr).map(incr).reduce(add)
```

and the following rule:

$$\texttt{map}(\texttt{map}(l, f), g) \to \texttt{map}(l, g \circ f)$$

where l, f, and g are variables.

The rule cannot be applied to the root of t, but the left-hand side of this rule matches the sub-term $t' = \texttt{map}(\texttt{map}(\texttt{pl}, \texttt{incr}), \texttt{incr})$ of t:

$$\begin{array}{c} match(\texttt{map}(\texttt{map}(\texttt{pl}, \texttt{incr}), \texttt{incr}),\quad \texttt{map}(\texttt{map}(l, f), g)) = \\ [l \mapsto \texttt{pl},\ f \mapsto \texttt{incr},\ g \mapsto \texttt{incr}] \end{array}$$

Applying the obtained substitution to the right-hand side of the rule gives:

$$\texttt{map}(\texttt{pl}, \texttt{incr} \circ \texttt{incr})$$

and replacing t' in t by this new term, we finally obtain:

$$\texttt{reduce}(\texttt{map}(\texttt{pl}, \texttt{incr} \circ \texttt{incr}), +)$$

This skeleton expression is likely to be more efficient because it avoids to create an intermediate list in-between the two calls to `map`. We use such rules in PySke to optimize skeleton compositions.

4.3 Term Rewriting in Python

Our framework first provides features that are close to the theoretical term rewriting system framework presented in the previous sub-section. These are provided in one Python module: `terms`.

The `terms` module offers two main classes: `Var` that just extends `str` and that corresponds to variables in the theoretical framework, and `Term` than implements non-variable terms. This latter class has two main attributes: `function` and `arguments` that basically correspond to the function symbol and arguments in a theoretical term.

We however take advantage of the flexibility of Python type system. The `function` attribute can be either a string, or a Python function. The arguments are represented as a list. Its elements can be either values of type `Term`, strings (in this case representing a class), or Python values of arbitrary types. The two latter cases can be thought as constants in the theoretical framework.

Substitutions in the theoretical framework are implemented as Python dictionaries. Applying a substitution to a term `t` (in Python a value either of type `Var` or of type `Term`) is implemented as a Python function `subst(t, s)` where `s` is a dictionary.

One important feature of the theoretical framework is the *match* function. It is implemented in Python as a method of the class `Term` that takes as argument a value either of type `Var` or of type `Term`. This function either returns a dictionary in case the matching succeeds, or `None` if the matching fails (`None` thus corresponds to $\perp$ of the theoretical framework).

Finally rules are implemented as a Python `namedtuple`, so essentially records. These records contain of course two terms, `left` and `right`, but also a `name` for the rule, and a `type`. This `type` is actually a sub-class of `Term` as we will explain in Section 4.4.

The module `terms` contains also a function to apply a rule at the root of a term, and a function `inner_most_strategy(t)` that repeatedly tries to apply all the available rules to every sub-terms of `t` until it is no longer possible to apply any rule. The strategy to apply the rules is discussed in the next section.

As an example, Figure 4 presents the encoding of the rule presented in the previous section. This example illustrates the fact that the `function` attribute of a term can be either a string (most of the cases) or a Python function. This is the case for `compose` (Line 4) which is a function defined as:

```python
def compose(f, g):
    return lambda x: f(g(x))
```

Similarly the elements of `arguments`, the second parameter to the constructor of `Term` can be either variables, terms, or arbitrary Python values (this latter case is not present in the example).

4.4 Rule-Based Skeleton Compositions Optimization

We have a framework to represent generic terms, and rules to transform such terms. There remain two main questions to use such a framework to automatically optimize skeleton compositions:

```
Rule(left=Term('map',
               [Term('map', [Var('PL'), Var('f')]), Var('g')]),
     right=Term('map',
                [Var('PL'), Term(compose,
                                 [Var('f'), Var('g')])]),
     name="map_map",
     type=_List)
```

Fig. 4: A Rewriting Rule in Python

1. How to represent PySke skeleton compositions as instances of `Term` in a way that is mostly transparent to the user with respect to the former PySke API?
2. How to evaluate/execute such terms?

The design principles for a solution to (1) are:

– inherit from `Term`, and
– use `Python` introspection capabilities to obtain a concise implementation.

Thus essentially, for each class that provided a data structure and associated skeletons in PySke we define a "wrapper" class that features the same methods than the initial class, but these methods build terms instead of executing parallel code.

A class such as `Plist` has static methods (and we identify the class constructor to a static method in our framework) and methods. For both kind of methods, the representation is such that:

– the method name is represented as the function symbol of the term (in class `Term`, it is attribute `function`),
– the class or object that is the target of the method call is represented as the *first* argument of the term (first element of the list `arguments`),
– the remaining of the method arguments are represented as additional elements in the list `arguments`.

Python offers many introspection features. In particular it is possible to capture any call to non-static methods that are not defined in a class. Therefore to define a `PList` wrapper class that inherits from `Term` we have basically to define static methods that correspond to the initial class static methods, and a generic way to catch calls to non-static methods to build the corresponding terms.

The initial `PList` class features the `init(f, size)` static method. The wrapper `PList(Term)` class thus features the following static method:

```
@staticmethod
def init(f, size):
    return PList('init', ['PList', f, size])
```

As indicated before, the value for the attribute `function` is a string that is the name of the corresponding static method in the initial PySke `PList` class, and the `arguments` list, in addition to the arguments to `init` that are `f` and `size`, contains as first element the name of the class (here `PList`) that contains the `init` static method.

As explained before, for non-static methods, we do not re-implement each method but rather rely on a feature of Python that allows to capture any call to non-defined methods in a class (technically we redefine the `__getattr__` attribute). We refer to the source code[3] for more technical details. An instance of this generic code for method `map` follows:

```
def map(self, f):
    return PList('map', [self, f])
```

Using this wrapper class `PList` is therefore exactly the same than using the initial `PList` but nothing is executed: instead a term representing the skeleton expression is built. The only difference are: the `raw` method that allows to embedded a value of the initial class into the wrapper class as illustrated in Section 4.1, and launching the optimization and execution of such a term.

Indeed, *in fine* it is necessary to execute the skeleton composition. This is done using a method called `run()`. It proceeds in two steps:

1. first the term is transformed using rules such as the rule presented in Figure 4,
2. then the optimized term is executed.

The optimization and execution features are not specifically implemented for each wrapper class. They are actually offered by two methods of the class `Term`:

- `opt()` transforms a term using all the available rules by calling the transformation function `inner_most_strategy` that applies all the available rules as many times as possible on all the sub-terms of the current term;
- `eval()` executes a term. We refer to the code for details, but basically Python allows to retrieve attributes of a class (including methods) by their names represented by a string, and to apply any function or method to a list of values in such a way that this list is considered as the effective parameters of the function or method. We use both this features to retrieve the target class and methods in the initial PySke classes from objects of the wrapper classes.

Strategy The transformation rules are applied using an innermost strategy: the rules are first applied (when possible) to the leaves of the tree representing the composition of skeletons, and then in an upwards fashion. When two rules may be applied to the same sub-expression of the skeleton expression, priorities associated to rules are used. As future work we plan to extend this default behavior as discussed in Section 6.

5 Applications and Experiments

Currently, our framework contains a dozen of optimization rules:

- optimization of a composition of `map`s,

[3]available at `https://github.com/pyske/PySke`

- optimization of compositions of `map` and `reduce` (using and internal `map_reduce` skeleton),
- optimization of compositions of `map` and `zip` to `map2`,
- optimization of compositions of `map` and `reduce` using boolean operators as their arguments (a generalized De Morgan rule),
- optimization of compositions of higher-order functions such as `uncurry` and `curry` (for example, the composition of these two functions is the identity function, and the identity function is a neutral element for function composition).

We evaluated our framework on three examples:

1. the dot product presented in Figure 3,
2. the conjunction of the negation of a list `l` of Boolean values:

   ```
   l.map(operator.not_).reduce(operator.and_, True)
   ```

3. the computation of the normalized average of vectors: a parallel list of vectors (with 10 components) is first normalized (the norm of each vector is computed and the vector is multiplied by a scalar that is the inverse of its norm), then the average of these lists is computed:

   ```
   smul(1 / l.length(), l.map(normalize).reduce(vadd, vzero))
   ```

The experiments were conducted on a shared memory machine (256 Gb), with two Intel Xeon E5-2683 v4 processors each having 16 cores at 2.10 GHz. Each example has been run 30 times. We used the following software: CentOS 7, Python 3.6.3, mpi4py 3.0.2, OpenMPI 2.6.4.

For the dot product example, we run four versions, on a parallel list of 5×10^7 elements:

- `direct` corresponds to `dot_product1`,
- `wrapper` builds the term representation of the skeleton composition, then executes it without transformation. This version allows to check the overhead introduced by the building of the intermediate representation and the overhead introduced by using such a representation for execution,
- `optimized` is the same than `wrapper` but the skeleton composition is optimized using all available rules,
- finally `hand_written` is the version using directly the `PList` but the composition is optimized manually.

The results are presented Figure 5. First of all, there is no noticeable overhead due to the wrapper classes with respect to the use of the direct implementation. That is true both for the non-optimized code (`direct` and `wrapper`) and the optimized code (`optimized` and `hand-optimized`). Secondly, the automatic optimization of PySke provides, in this case, a significant speedup of about 2.5.

For the list of Boolean values example, the average speed-up of the automatically optimized version with respect to the direct implementation is 85%. For the vector average after normalization, the average speed-up is more modest at 16%.

In all cases, the speed-up is there for only a slight change in the program with respect to the former API.

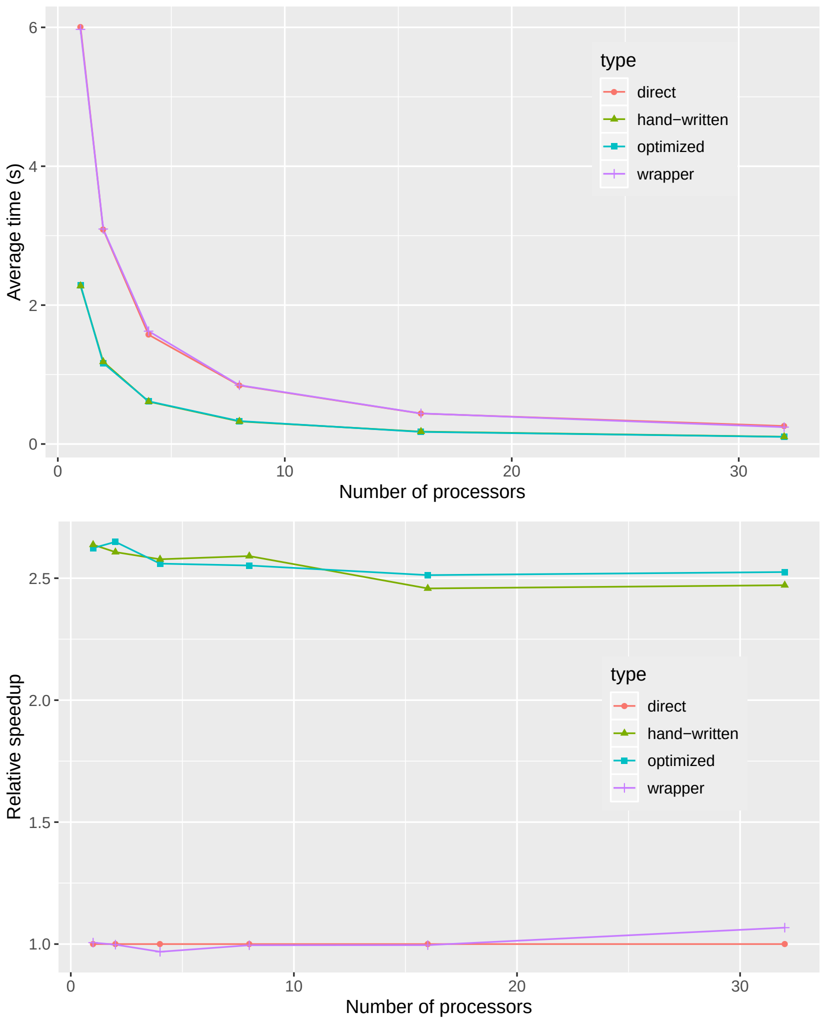

Fig. 5: Dot Product Example

6 Conclusion and Future Work

Skeletal parallel programming is a productive way to write parallel programs. With PySke we propose a library simple to use for non-expert programmers. However even with algorithmic skeletons, two difficulties remain: the choice of skeletons when the library is large, and the way these skeletons are composed to form an application. These choices have a large impact on the performance of the parallel program. To further eases the development of efficient parallel programs with PySke, we propose automatic program transformation based on high-level transformation rules. Evaluation of this new mechanism in PySke shows it can provide significant improvements to skeletal programs.

For the moment the transformation rules are hand-written by the developers of PySke. We plan to have a more flexible approach by extending our framework so it can read transformation rules from a file. This would allow expert users of PySke to add more domain specific transformation rules. The transformation rules will be written in a format similar to what can be found in term rewriting tools such as ELAN [9]. Another extension is to provide ways to change the strategy of application of these rules. A very flexible solution would be to allow expert users to write their own strategies as it is possible in embedded term rewriting tools [5].

References

1. Aldinucci, M., Danelutto, M., Teti, P.: An Advanced Environment Supporting Structured Parallel Programming in Java. Future Generation Computer Systems **19**, 611–626 (2002)
2. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: the FAN skeleton framework. Parallel Algorithms Appl. **16**(2-3), 87–121 (2001). https://doi.org/10.1080/01495730108935268
3. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, New York (1998)
4. Bacci, B., Gorlatch, S., Lengauer, C., Pelagatti, S.: Skeletons and transformations in an integrated parallel programming environment*. In: Malyshkin, V. (ed.) Parallel Computing Technologies. pp. 13–27. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
5. Balland, E., Moreau, P., Reilles, A.: Effective strategic programming for Java developers. Softw., Pract. Exper. **44**(2), 129–162 (2014). https://doi.org/10.1002/spe.2159
6. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall (1996)
7. Bird, R.S.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design. pp. 5–42. Springer Berlin Heidelberg, Berlin, Heidelberg (1987). https://doi.org/10.1007/978-3-642-87374-4_1
8. Bougé, L.: The data parallel programming model: A semantic perspective, pp. 4–26. Springer Berlin Heidelberg, Berlin, Heidelberg (1996). https://doi.org/10.1007/3-540-61736-1_40
9. van den Brand, M., Moreau, P., Ringeissen, C.: The ELAN environment: a rewriting logic environment based on ASF+SDF technology - system demonstration. Electr. Notes Theor. Comput. Sci. **65**(3), 50–56 (2002). https://doi.org/10.1016/S1571-0661(04)80426-2
10. Chakravarty, M.M.T., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating Haskell array codes with multicore GPUs. In: Workshop on Declarative Aspects of Multicore Programming (DAMP). pp. 3–14 (2011). https://doi.org/10.1145/1926354.1926358

11. Ciechanowicz, P., Poldner, M., Kuchen, H.: The Münster Skeleton Library Muesli – A Comprenhensive Overview. Tech. Rep. Working Paper No. 7, European Research Center for Information Systems, University of Münster, Germany (2009)
12. Clifton-Everest, R., McDonell, T.L., Chakravarty, M.M., Keller, G.: Streaming irregular arrays. In: Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. pp. 174–185. ACM (2017). https://doi.org/10.1145/3122955.3122971
13. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989)
14. Dalcin, L.D., Paz, R.R., Kler, P.A., Cosimo, A.: Parallel distributed computing using Python. Advances in Water Resources **34**(9), 1124 – 1139 (2011). https://doi.org/https://doi.org/10.1016/j.advwatres.2011.04.013, new Computational Methods and Software Tools
15. Emoto, K., Matsuzaki, K.: An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo. Int J Parallel Prog (2013). https://doi.org/10.1007/s10766-013-0263-8
16. Enmyren, J., Kessler, C.: SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. In: 4th workshop on High-Level Parallel Programming and Applications (HLPP). ACM (2010)
17. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: Quaff: Efficient C++ Design for Parallel Skeletons. Parallel Computing **32**, 604–615 (2006)
18. Légaux, J., Loulergue, F., Jubertie, S.: Managing Arbitrary Distributions of Arrays in Orléans Skeleton Library. In: International Conference on High Performance Computing and Simulation (HPCS). pp. 437–444. IEEE, Helsinki, Finland (2013). https://doi.org/10.1109/HPCSim.2013.6641451
19. Pelagatti, S.: Structured Development of Parallel Programs. Taylor & Francis (1998)
20. Philippe, J., Loulergue, F.: PySke: Algorithmic skeletons for Python. In: International Conference on High Performance Computing and Simulation (HPCS). pp. 40–47. IEEE, Dublin, Ireland (2019)
21. Philippe, J., Loulergue, F.: Towards automatically optimizing PySke programs (poster). In: International Conference on High Performance Computing and Simulation (HPCS). pp. 1045–1046. IEEE, Dublin, Ireland (2019)
22. Sujeeth, A.K., Brown, K.J., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: Delite: A compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans. Embed. Comput. Syst. **13**, 134:1–134:25 (Apr 2014). https://doi.org/10.1145/2584665