New List Skeletons for the Python Skeleton Library

Frédéric Loulergue, Jolan Philippe

▶ **To cite this version:**

Frédéric Loulergue, Jolan Philippe. New List Skeletons for the Python Skeleton Library. PDCAT 2019: 20th International Conference on Parallel and Distributed Computing, Applications and Technologies, Dec 2019, Gold Coast, Australia. 10.1109/PDCAT46702.2019.00077 . hal-02317124

HAL Id: hal-02317124
https://hal.science/hal-02317124

Submitted on 15 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

New List Skeletons for the Python Skeleton Library

Frédéric Loulergue*†, Jolan Philippe‡§
*School of Informatics Computing and Cyber Systems, Northern Arizona University, USA
†IMT Atlantique, Inria, LS2N, UMR CNRS 6004, F-44307 Nantes, France
‡frederic.loulergue@nau.edu, §jolan.philippe@imt-atlantique.fr

***Abstract*—Algorithmic skeletons are patterns of parallel computations. Skeletal parallel programming eases parallel programming: a program is merely a composition of such patterns. Data-parallel skeletons operate on parallel data-structures that have often sequential counterparts. In algorithmic skeleton approaches that offer a global view of programs, a parallel program has therefore a structure similar to a sequential program but operates on parallel data-structures. PySke is such an algorithmic skeleton library for Python to program shared or distributed memory parallel architectures in a simple way. This paper presents an extension to PySke: new algorithmic skeletons on parallel lists. This extension is evaluated on an application.**

***Index Terms*—high-level parallel programming, structured parallelism, algorithmic skeleton, Python, lists.**

I. Introduction

Parallel architectures are now everywhere, from smartphones to super-computers. But while parallel architectures are mainstream, it is not the case for parallel programming and skilled parallel programmers. Python is a language more and more popular: in popularity rankings (based on internet search queries) and in lines of code in public repositories it is ranked as one of the first three programming languages in 2019. It is also more and more popular as an introductory teaching language, both in computer science curricula [1] but also in informatics curricula. A recent survey of the Python Software Foundation and JetBrains [1] also shows that in 2018 58% of developers used Python for data analysis (from 50% in 2017). This ratio is even higher for the 84% of developers who use Python as their main programming language.

As the size of data to analyze grows, it is necessary to provide the programmers who are not expert in parallel programming, and often not computer scientists, ways to leverage the parallel architectures they have for data analysis in Python.

In the various approaches for high-level parallel programming, algorithmic skeletons [2], [3], [4] are quite popular even though the most popular frameworks belonging to this family of approaches are not presented as skeletal parallelism frameworks. Basically, an algorithmic skeleton is a pattern of parallel computation. There are task- and data-parallel skeletons. Often data-parallel skeletons operate on parallel data-structures that have sequential counterparts, and therefore the skeletons themselves have sequential counterparts. For example the map skeleton is often a skeleton that applies a sequential function to all the elements of a parallel collection and it is also a function on sequential lists in functional programming languages. The most popular skeleton framework is Google's MapReduce [5] although the skeletons it provides differ from more traditional map and reduce skeletons.

PySke [6] is a library of algorithmic skeletons for Python, on parallel lists and parallel trees. PySke provides a global view of programs, i.e. a PySke program is written as a sequential program but it operates on parallel data-structures. It is simpler than the Single Program Multiple Data (SPMD) style of most MPI programs. When possible (it is not possible for all skeletons on trees), PySke provides the same methods for both a sequential and a parallel data-structure.

The contributions of the present paper are:

- new expressive algorithmic skeletons on lists,
- evaluation of the new skeletons on applications.

The remaining of the paper is organized as follows. We discuss related work in Section V. Section II gives an overview of PySke both from the user and implementer perspectives. In Section III, we present new skeletons on the parallel list data-structure. We experiment with these new skeletons in Section IV and conclude in Section VI.

II. An Overview of PySke

A. A User's Perspective

PySke is a library for Python currently implemented on top of MPI and *mpi4py* [7]. However, PySke's programming model is independent of the communication library.

PySke offers a *global view* of programs. A PySke program is written and read as a sequential program but 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). The global parallel MPI 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. Nevertheless there is a major difference between MPI and PySke. Arguments to MPI collectives have regular C types but the collection all 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 sequential/parallel nature of values is

[1] https://www.jetbrains.com/research/python-developers-survey-2018

```
def _max0_copy(num):
    return max(0, num), num

def _max_sum(pair_a, pair_b):
    a_m, a_s = pair_a
    b_m, b_s = pair_b
    max_ = max(a_m, a_s + b_m)
    sum_ = a_s + b_s
    return max_, sum_

def mps(input_list: List):
    max_, _ = input_list
                .map(_max0_copy)
                .reduce(_max_sum, (0, 0))
    return max_
```

Fig. 1. A First PySke Example

therefore visible in their types. No parallel type exists in MPI, and it may be difficult (and it is an undecidable problem in general) to know whether a value is sequential or should be thought as being part of a distributed data structure.

PySke contains two main kinds of data structures: lists and trees. We refer to [6] for a presentation of the tree data structures. In the present paper, we focus on list data structures. With respect to the first version of PySke, we know have an interface `List` that contains the documentation for all the methods on lists. It also features detailed type hints that we consider to be very useful documentation items. Both **`SList`**, the class of sequential lists, and **`PList`**, the class of parallel lists, implement this interface. In the remaining, a method on a sequential data structure is called a *primitive*, while it is called an *algorithmic skeleton*, or *skeleton*, on a parallel data structure. It is important to notice that **`PList`** are *immutable* data structures. All skeletons and primitives return new lists or scalar values. For each primitive we provide a skeleton, and vice versa. This allows for the rapid prototype of PySke applications, even without mpi4py installed. While **`SList`** is strictly speaking not an immutable data structure, because it inherits from `list` for convenience, the recommended style is to use it as an immutable data structure.

The program example of Figure 1 illustrates the polymorphic nature of PySke. The variable `input_list` can be either **`SList`** or **`PList`**. In the former case, the execution can be sequential only. In the later case, it uses MPI for parallel processing.

`mps` returns the largest sum of all the sums of the prefixes of the input list. For example, the prefix list returning the largest sum, is `[6, -3, 0, 4]` in the following execution[2]:

```
>>> mps(PList.from_seq([6, -3, 0, 4, -4]))
7
```

`from_seq` builds a parallel list from a sequential list.

[2]We present the code and results as given in the Python interpreter, omitting imports. **`>>>`** represents the prompt.

Global View $[0, 2, 4, 6, 8, 10, 12, 14, 16]$				
SPMD View				
processor	0	1	2	3
content	$[0, 2, 4]$	$[6, 8]$	$[10, 12]$	$[14, 16]$
global size	9	9	9	9
local size	3	2	2	2
start index	0	3	5	7
distribution	$[3, 2, 2, 2]$	$[3, 2, 2, 2]$	$[3, 2, 2, 2]$	$[3, 2, 2, 2]$

Fig. 2. Parallel Lists: API View vs. Implementation View

`mps` is implemented as a composition of **`map`** and **`reduce`**. Simpler examples of application of both skeletons follows:

```
>>> SList([1, 2, 3]).map(lambda x: x + 1)
[2, 3, 4]
>>> SList([1, 2, 3]).reduce(add, 0)
6
>>> SList([]).reduce(add, 0)
0
```

Basically, to compute the maximum prefix sum, we compute a pair containing the maximum prefix sum and the sum. The intuition behind the definition of `_max0_copy` is that if we consider an element as a singleton list. The maximum prefix sum for this list is either 0 if the element is negative (the prefix is then the empty list), or the element itself. `_max_sum` is defined as follows: `pair_a` represents the maximum prefix sum (`a_m`) and the sum (`a_s`) of a list `a`, and `pair_b` the same for a list `b`. To compute the maximum prefix sum of the list obtained by concatenating `a` and `b`, we need to take the maximum of the maximum prefix sum for `a` and the maximum prefix for `a` followed by `b`. The maximum prefix sum value for this second list is `a_s + b_m`.

There are several variants of the **`map`** skeleton:

1) **`mapi`** where the function argument to **`mapi`** is applied to each index and associated value in the list,
2) **`map2`** that takes two arguments in addition to the current object: a function and a list, and applies the function to elements of both lists; these two lists should have the same distribution (as explained in the next section); **`map2i`** is a variant of **`map2`**,
3) **`zip`** creates a list of pairs from two lists having the same distribution.

There are several variants of the prefix sum skeleton **`scanl`**. An example of **`scanl`** follows:

```
>>> par_list = PList.init(str, 4)
>>> par_list.scanl(concat, '').to_seq()
['', '0', '01', '012']
```

`init` is the main way to build parallel list in PySke. It takes as argument a function used to initialize the values of the list, and the size of the list. **`to_seq`**`()` transforms a parallel list into a sequential list. `par_list.`**`to_seq`**`()` is `['0', '1', '2', '3']`.

B. An Implementer's Perspective

Users can think about their **PList** in a global way, i.e. as a sequential list of type **SList**. However these lists can be indeed distributed on different machines, in a transparent machine. The underlying implementation of PySke follows the SPMD paradigm and is currently implemented using MPI.

For an example list, Figure 2 shows both the global view presented to the user of the library, and the way this view is realized following the SPMD paradigm (for 4 processors). Each row in the SPMD view corresponds to a private attribute in the Python implementation. Names in italic represent attributes that contain the same values on all processors. "content" is specific to each processor and contains the specific part of the global list that this given processor has. Note that the type of this attribute in **SList**. "start index" is the index in the global size of the first element of the local list. In the global list, the value 6 has index 3. But this value is the first element of the local list at processor 1. Therefore the value of "start index" at processor 1 is 3. The "distribution" attribute contains the list of all local sizes.

init builds an evenly distributed list, i.e. a list where the local sizes differ at most by 1, and the possibly larger local lists are on processors of low identifier. The list presented in Figure 2 is such an evenly distributed list. **from_seq** does not build an evenly distributed list: all the elements at are processor 0.

map, **scanl** and their variants return lists that have the same distribution as their input. When a PySke skeleton manipulates more that one list, all these lists should have the same distribution.

C. Distribution Changing Skeletons

Some skeletons exist to change the distribution of a list. **get_partition** makes the distribution of the lists visible in the structure itself. For example, **get_partition** on the list of type **PList** in Figure 2 yields the **PList** $[[0, 2, 4], [6, 8], [10, 12], [14, 16]]$ (global view). Now each processor contains only one element (local size 1), but this element is a list. The distribution of this new list is $[1, 1, 1, 1]$.

flatten is the inverse operation of **get_partition**. Communications are needed to compute the new values of all the attributes but "content".

The **filter** skeleton that filters out elements in a list that does not satisfy the given predicate can easily be implemented using these skeletons:

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

Note that **filter** is not recursive: this code is the method of **PList**, while in the application of **map**, **filter** is the sequential filter of class **SList**. Also the distribution of the list returned by **filter** may be unbalanced: all the elements of one processor may have been filtered out while the other processors keep their contents. One very important feature of **filter** is that it is implemented only using other PySke skeletons. We call such skeletons, *derived skeletons*. They are skeletons that PySke users could define. Most other skeleton libraries require skeletons such as **filter** to be defined at the implementer's level. Having a set of skeletons expressive enough to be able to define derived skeletons makes PySke more maintainable than other libraries. It also allows to have intermediate level users who do not need to know the implementation details of PySke but can still provide derived skeletons to less skilled users.

An example of use follows:

```
>>> par_list = PList.init(lambda x: x, 10)
>>> par_list.filter(is_even).to_seq()
[0, 2, 4, 6, 8]
```

The **balance** skeleton simply makes a list evenly distributed. This skeleton perform communications to exchange content between processors.

III. New List Skeletons

This section present new list skeletons for PySke.

A. The distribute Skeleton

The **distribute** skeleton is a generalization of the **balance** skeleton. The **balance** skeleton assumes that the target distribution is the default distribution when parallel lists are created: each processor contains at most one more value than each other processor. However, such a distribution may not be appropriate.

First, the size of the elements in the parallel list may be different (for example if we have a list of matrices of different sizes) and/or the computational resources needed to deal with the elements in the list depends on the values or on the size of each element. In this case a non evenly distribution of data may lead to more balanced computations.

Second, some skeletons that are applied to both the current parallel list object and an additional parallel list object require that the distributions of the parallel lists are identical. Rather than balancing both parallel list, it may be more efficient to redistribute one of the lists so its distribution matches the distribution of the other list.

The skeleton **distribute** takes an argument: a sequential list that represents the target distribution. This list should have a length equal to the number of processors, the values contained in this list should be positive (possibly zero), and the sum of these values should be equal to the global size of the parallel list the **distribute** method is applied to.

This skeleton proceeds as follows: first it computes the global indices of the first and last element for each processor, both in the source distribution and in the target distribution. This gives either an interval of indices, or the empty interval $\emptyset$. Then each processor intersects its source global interval with the target global intervals of all processors (including itself). These global intervals are offset to local intervals using the start index of the source parallel list. Finally these intervals are used to get the slices of the local contents of the parallel list to send to other processors. After a call to the MPI

Source start index	$[0, \perp, 10, 20]$
Source distribution	$[10, 0, 10, 15]$
Target distribution	$[5, 10, 10, 10]$
Source global intervals	$[(0, 9), \emptyset, (10, 19), (20, 34)]$
Target global intervals	$[(0, 4), (5, 14), (15, 24), (25, 34)]$
Global intervals to send	$[(0, 4), (5, 9), \emptyset, \emptyset]$ $[\emptyset, \emptyset, \emptyset, \emptyset]$ $[\emptyset, (10, 14), (15, 19), \emptyset]$ $[\emptyset, \emptyset, (20, 24), (25, 34)]$
Local intervals to send	$[(0, 4), (5, 9), \emptyset, \emptyset]$ $[\emptyset, \emptyset, \emptyset, \emptyset]$ $[\emptyset, (0, 4), (5, 9), \emptyset]$ $[\emptyset, \emptyset, (0, 4), (5, 14)]$

Fig. 3. Example of Execution of the distribute Skeleton

`alltoall` function, the resulting list of lists is flattened to a list. This gives the content of the local lists, the other fields of the parallel list are easily computed. The global size is the global size of the input parallel list. The distribution and the local sizes are given by the argument to **`distribute`**. The start index is obtained by computing the **`scanl`** of the target distribution. Figure 3 shows some of the intermediate values for an example parallel list of size 15 on 4 processors.

B. New Derived Skeletons

The **`distribute`** skeleton allows to implement communication oriented derived skeletons as shown in Figure III-B. Such skeletons are often present in other libraries, but there are implemented at a low level: in SPMD using directly MPI calls. In our case, the implementation of these derived skeletons could have been written by a PySke user without knowing the implementation details of PySke. This has two main advantages:

- the implementation is more maintainable,
- the PySke implementation is also more portable. As we mentioned earlier, the programming model of PySke is not tied to MPI. For example, a shared memory implementation of PySke is possible. With a high-level implementation of **`scatter`** and **`gather`**, it is not necessary to re-implement them for the shared memory version of PySke. The implementations of Figure III-B would work with any new implementation of the non-derived algorithmic skeletons.

Informally, **`gather`** gathers all the elements of the current list to the given processor. **`scatter`** scatters all the elements hold by the specified processor to all processors. The returned list is evenly distributed. **`scatter_range`** is a variant of **`scatter`**. Instead of specify a processor identifier as the source of the elements to scatter, these elements are specified by their indices in the list, as a range.

The code of Figure III-B only uses already described skeletons. The distribution information is actually defined as an object of `Distribution`. This class inherits from 'list', and it provides the static method `balanced` that creates an evenly distributed distribution for a list of the given size.

```
def gather(self, pid):
    assert pid in par.procs()
    d_list = [self.length() if i == pid
              else 0 for i in par.procs()]
    distr = Distribution(d_list)
    return self.distribute(distr)

def scatter(self, pid):
    assert pid in par.procs()
    def select(index, a_list):
        if index == pid:
            return a_list
        return []
    at_pid = self.get_partition()
                 .mapi(select).flatten()
    size = at_pid().length()
    distr = Distribution.balanced(size)
    return at_pid.distribute(distr)

def scatter_range(self, rng):
    def select(index, value):
        if index in rng:
            return value
        return None
    def not_none(value):
        return value is not None
    selected = self.mapi(select)
                   .filter(not_none)
    size = selected.length()
    distr = Distribution.balanced(size)
    return selected.distribute(distr)
```

Fig. 4. Derived Algorithmic Skeletons

`par.procs()` is not a skeleton but is part of a small set (in module `par`) of utility functions. `procs()` simply returns the list of available processors.

C. The permute Skeleton

The **`permute`** skeleton aims at reordering the elements of a list without changing its distribution. It takes as argument a bijective function on the set of the list indices.

For example:

```
>>> SList([1, 2, 3]).permute(lambda x: 2-x)
[3, 2, 1]
```

The implementation of **`permute`** on **`PList`** proceeds as follows: First, each element in the local lists are tupled with their new index in the global list, and the processor that holds this index, Then, all these tuples are then grouped by processor of destination. The obtained lists of pairs (new index, value) are exchanged using MPI `alltoall`. Finally, the local contents are updated using these lists of pairs.

The steps manipulating lists are implemented using **`SList`** primitives.

```
1  def fft(input_list: PList[float]) -> PList[complex]:
2      size, nprocs = len(input_list), len(par.procs())
3      log2_s, log2_p = int(math.log2(size)), int(math.log2(nprocs))
4      result = input_list.map(complex)
5      for j in range(0, log2_p):
6          permutation = result.get_partition()\
7              .permute(partial(_bit_complement, log2_p - j - 1))\
8              .flatten()
9          result = permutation.map2i(partial(_combine, size, log2_s, j), result)
10     for j in range(log2_p, log2_s):
11         permutation = result.get_partition()\
12             .map(lambda l: l.permute(partial(_bit_complement, log2_s - j - 1)))\
13             .flatten()
14         result = permutation.map2i(partial(_combine, size, log2_s j), result)
15     return result
```

Fig. 5. Fast Fourier Transform

To compute the processor that holds an index, it is first necessary to compute the `scanr` variant of the prefix sum of the distribution of the parallel list. In this variant, the result list has the same length as the input list, but its first element is the first element of the initial list. `scanr` takes only one argument in addition to the input list: a binary operation.

For example, if we consider the list of Figure 2, the prefix sum `scanr` returns the list $[3, 5, 7, 9]$. Then for a given index i, we search the value v in this list such that i is smaller than v and i is greater or equal to the value preceding v in the list. For example, for index 6, this value is 7. The index of 7 in the list being 2, the processor holding the index 6 is processor 2.

IV. Experiments

We experiment on a PySke parallel implementation of the Fast Fourier Transform (Figure 5). The algorithm needs $\log n$ steps to complete, where n is the size of the input list. Both the number of processors and n should be a power of two. Basically, the computation needs to combine successively elements at indexes that correspond to a butterfly network. As the list is distributed on processors, the algorithm has therefore two main phases. In the first phase (lines 5–9), it is necessary to permute whole partitions of the current parallel list. In the second phase (lines 10–14), it is necessary to permute elements only in the same local partition. As PySke provides the same skeletons for both parallel lists and sequential lists, and that in a parallel list the local partitions are sequential lists, both phases are actually very similar. `_bit_complement` computes the correct indices that correspond to the butterfly network. `_combine` actually computes the new complex number based on two complex numbers that are linked in the butterfly network, and a complex number on the unit circle that only depends on the size of the input list, and the indexes of both complex numbers.

The experiments have be conducted on a shared memory machine with two Intel Xeon E5-2683 v4 processors each having 16 cores at 2.10 GHz, and a memory of 256 GB. The machine was running CentOS 7. We used Python 3.6.3, OpenMPI 4.0.1, and mpi4py 3.0.2. Each point on Figure 6 is the mean of a series of 30 measures. The size of the input list was 2^{18} floating point numbers. The speed-up is a relative speed-up with respect to the timing for 1 processor. PySke achieves a good relative speed-up in this case.

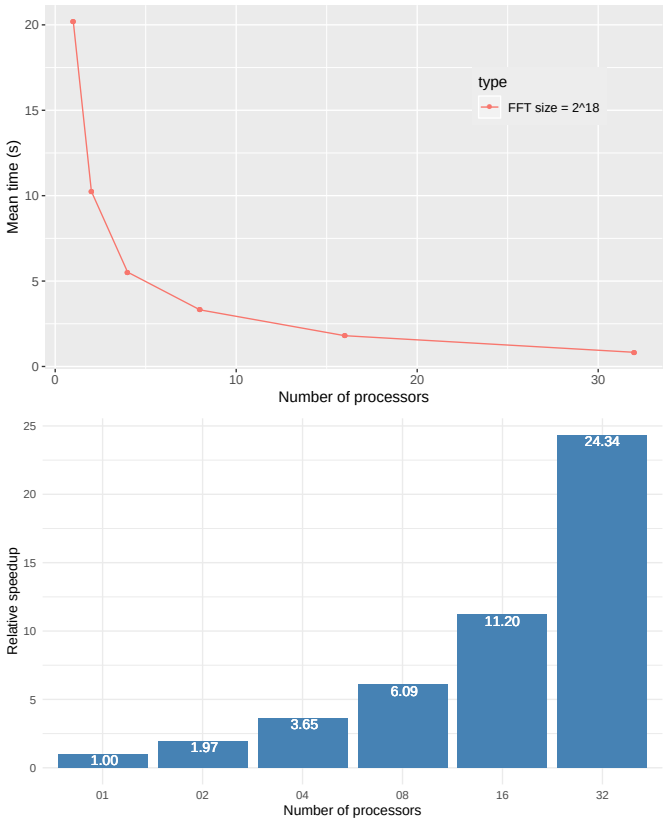

Fig. 6. FFT Timing and Speed-up

V. Related work

Algorithmic skeletons are, of course, naturally combined with functional languages. In this context, an algorithmic skeleton is simply a higher-order function implemented in parallel. There are several skeleton approaches available for functional programming languages. For OCaml, OCamlP3L [9] and its successor Sklml offer a set of a small set data and task parallel skeletons. Both rely on imperative features of OCaml. parmap [10] is a lightweight skeleton library that provides only parallel map and reduce on shared memory machines. Algorithmic skeletons for OCaml have also been implemented using the Bulk Synchronous Parallel ML library [11]. The proposed library is similar to PySke, but PySke has a much richer set of skeletons. For Haskell, Accelerate is a skeleton library that targets GPUs only. The initial proposal [12] featured classical data parallel skeletons (map and variants, reduce, scan and permutation skeletons) on multi-dimensional arrays. Both these languages are statically typed languages, more difficult to learn, and less used than Python.

PySke is related to SkeTo [13], [14], OSL [15], [16], and Muesli [17], [18], [19]. All these libraries are C++ libraries. They are, of course, more efficient than PySke, but using them is less flexible, and requires a higher level of expertise. PySke contains all the skeletons of OSL but the BSP homomorphism skeleton [15] and the exception forwarding skeleton [20]. However, now that base infrastructure of PySke is solid, adding these two skeletons will not be a complex task. PySke contains additional skeletons on lists with respect to OSL.

Muesli provides a `permutePartition` skeleton on arrays that is limited to permuting whole partitions. The **`permute`** skeleton of PySke is more fine grained but used in combination with **`get_partition`**, it can implement the same behavior than Muesli's `permutePartition` as shown by the FFT example. Muesli also offers gathering and scattering skeletons: they are not derived skeletons as our **`scatter`** and **`gather`** skeletons are.

Both compared to OSL and Muesli, PySke has the advantage of providing skeletons on trees. This is a feature shared with earlier versions of SkeTo, but not with recent ones. PySke is therefore unique in this respect. SkeTo and Muesli provide skeletons on 2D matrices in addition to 1D arrays.

VI. Conclusion and Future Work

Programming with algorithmic skeletons is a productive way of writing parallel programs. PySke is a simple library for non-expert programmers, but thanks to its rich set of skeletons, it is very expressive. Experiments shows performances are scalable.

We plan to continue adding new skeletons for the existing PySke data-structures, and new data structures such as multi-dimensional arrays.

As the set of skeletons offered by PySke grows, PySke users may find it difficult to choose the right skeleton, and to rightly compose them. In order to ease the work of PySke users, we plan to design and implement a mechanism for skeleton composition optimization by program transformation. This mechanism will be based on concepts of term rewriting systems [21], and will be performed at runtime. Preliminary results on non-automatic transformations showed the potential of such an approach [6].

References

[1] P. Guo, "Python is now the most popular introductory teaching language at top u.s. universities," BLOG@CACM, July 2014. [Online]. Available: https://cacm.acm.org/blogs/blog-cacm/176450

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

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

[4] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Software, Practrice & Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*. USENIX Association, 2004, pp. 137–150.

[6] J. Philippe and F. Loulergue, "PySke: Algorithmic skeletons for Python," in *International Conference on High Performance Computing and Simulation (HPCS)*. Dublin, Ireland: IEEE, 2019.

[7] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo, "Parallel distributed computing using Python," *Advances in Water Resources*, vol. 34, no. 9, pp. 1124 – 1139, 2011, new Computational Methods and Software Tools.

[8] L. Bougé, *The data parallel programming model: A semantic perspective*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 4–26.

[9] R. D. Cosmo, Z. Li, S. Pelagatti, and P. Weis, "Skeletal Parallel Programming with OcamlP3l 2.0," *Parallel Processing Letters*, vol. 18, no. 1, pp. 149–164, 2008.

[10] R. Di Cosmo and M. Danelutto, "A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml," in *International Conference on Computational Science (ICCS)*, vol. 9. Elsevier, 2012, pp. 1837–1846.

[11] F. Loulergue, "Implementing Algorithmic Skeletons with Bulk Synchronous Parallel ML," in *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 2017, pp. 461–468.

[12] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating Haskell array codes with multicore GPUs," in *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011*, 2011, pp. 3–14.

[13] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, "A Library of Constructive Skeletons for Sequential Style of Parallel Programming," in *InfoScale'06: Proceedings of the 1st international conference on scalable information systems*. ACM, 2006.

[14] K. Emoto and K. Matsuzaki, "An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo," *Int J Parallel Prog*, 2013.

[15] J. Légaux, F. Loulergue, and S. Jubertie, "Managing Arbitrary Distributions of Arrays in Orléans Skeleton Library," in *International Conference on High Performance Computing and Simulation (HPCS)*. Helsinki, Finland: IEEE, 2013, pp. 437–444.

[16] J. Légaux, S. Jubertie, and F. Loulergue, "Development Effort and Performance Trade-off in High-Level Parallel Programming," in *International Conference on High Performance Computing and Simulation (HPCS)*. Bologna, Italy: IEEE, 2014, pp. 162–169.

[17] P. Ciechanowicz, M. Poldner, and H. Kuchen, "The Münster Skeleton Library Muesli – A Comprenhensive Overview," European Research Center for Information Systems, University of Münster, Germany, Tech. Rep. Working Paper No. 7, 2009.

[18] P. Ciechanowicz and H. Kuchen, "Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures," in *IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2010, pp. 108–113.

[19] F. Wrede, B. A. De Melo Menezes, and H. Kuchen, "Fish school search with algorithmic skeletons," *International Journal of Parallel Programming*, vol. 47, no. 2, pp. 234–252, 2019.

[20] J. Légaux, F. Loulergue, and S. Jubertie, "OSL: an algorithmic skeleton library with exceptions," in *International Conference on Computational Science (ICCS)*. Barcelona, Spain: Elsevier, 2013, pp. 260–269.

[21] Terese, *Term Rewriting Systems*. Cambridge University Press, 2003.