PySke: Algorithmic Skeletons for Python

Jolan Philippe
School of Informatics Computing and Cyber Systems
Northern Arizona University
Flagstaff, AZ, USA
jp2589@nau.edu

Frédéric Loulergue
School of Informatics Computing and Cyber Systems
Northern Arizona University
Flagstaff, AZ, USA
frederic.Loulergue@nau.edu

***Abstract*—PySke is a library of parallel algorithmic skeletons in Python designed for list and tree data structures. Such algorithmic skeletons are high-order functions implemented in parallel. An application developed with PySke is a composition of skeletons. To ease the write of parallel programs, PySke does not follow the Single Program Multiple Data (SPMD) paradigm but offers a global view of parallel programs to users. This approach aims at writing scalable programs easily. In addition to the library, we present experiments performed on a high-performance computing cluster (distributed memory) on a set of example applications developed with PySke.**

***Keywords*—High-level parallel programming, algorithmic skeletons, distributed lists, distributed trees, Python**

I. Introduction

a) Context and Motivation: The size of data manipulated by applications increases. Therefore, considering parallel programming is a necessity. Most devices are now using parallel architectures. However, parallel programming used to be only handled by high-performance computer scientists and data analysts and was a niche area. Nowadays, it is entirely mainstream. Through language libraries, developers can take advantage of the full resources of parallel machines.

There are two main approaches: shared-memory computing and distributed-memory computing. Some hybrid architectures use both. Shared-memory computing is usually used for small parallel computations because the processing units are manipulating the same data and then it does not necessitate inter-processors communications. With a distributed memory architecture, each processor has its private memory, and the exchange of information must be operated by concrete communications.

Parallel programming with distributed memory has been shown very efficient to treat large size data but remains challenging for developers. Indeed, several aspects must be considered such as communications, synchronization, and load balancing. The consequence of these difficulties is a lack of parallel programmers.

There exist compilers that can automatically parallelize sequential programs. But because compilers must be designed for a vast spectrum of cases, the resulting parallel programs can lose efficiency compared to hand-made parallel programs. In this paper, we aim at enabling the writing of efficient parallel programs using techniques that do not hinder programming productivity.

b) Approach and Goals: To ease the development of parallel programming, Murray Cole has introduced the skeletal parallelism approach [1]. A skeleton is a parallel implementation of a computation pattern. Using this approach, a developer does not have to think about parallelization anymore but only about how to write a program using these specific patterns. There exist skeleton libraries in high-level programming languages. For example, in C++, we have libraries such as SkeTo [2], SkePu [3], Muesli [4] or OSL [5].

The advantage of Python is its flexibility, and the multiple paradigms it provides that ease the development of applications. For instance, next to the object-oriented programming style, we can use lambda expressions. Lambda expressions and then higher-order functions are part of the language. Since skeletons are based on computation patterns that generally take functions as parameters, these functional notions are very convenient in the development of a skeleton library. The goal of PySke is to provide a straightforward way to write parallel programs using skeletons and without taking care explicitly of the parallel aspects of a program.

c) Contributions: Most of the already defined skeleton libraries does not provide skeletons for more than one data structure. Besides, they are mostly implemented in C++, Java or less mainstream functional languages. Fortunately, the Message Passing Interface (MPI) standard library has been ported to Python through the *mpi4py* [6]–[8] library. Using this package, we have developed Python skeletons for two data-structures, lists and trees, following an object-oriented programming (OOP) style. These structures are often involved in solving data analysis problems. Combining the OOP style, and the expressivity of Python, it is very comfortable to write parallel programs with PySke. To illustrate the scalability of the skeletons implementation, we conducted tests with examples based on problems solvable with these two data structures: the variance of a random variable, and the numbering of the elements of a tree based on a prefix traversal.

d) Outline: The organization of the paper is as follows. We first present related work Section II. Section III and IV present the implementation of skeletons on lists and arbitrary-shaped trees respectively. Section V is devoted to experiments on applications developed using these skeletons. We finally conclude and discuss future work in Section VI.

II. Related Work

Algorithmic skeletons were originally inspired by functional programming. It is not a surprise that several functional programming languages have algorithmic skeleton libraries. For OCaml, OCamlP3L [9], and its successor Sklml, offer a set of a few data and task parallel skeletons. Both rely on imperative features of OCaml. parmap [10], a lightweight skeleton library, provides only parallel map and reduce on shared memory machines. BSML [11], a bulk synchronous functional parallel programming library, is also used to implementation algorithmic skeleton libraries for OCaml [12]. All these libraries only operate on arrays and lists, not trees. While PySke does not provide task parallelism skeletons, its set of skeletons on lists is richer than the set of data parallel skeletons of the other libraries.

Eden is a non-pure extension to the Haskell language [13] that is also used to implement higher-level skeletons [14]. Accelerate is a skeleton library for Haskell that targets GPUs only. The initial proposal [15] featured classical data parallel skeletons (map and variants, reduce, scan and permutation skeletons) on multi-dimensional arrays. Besides, following an algorithmic skeleton approach, Accelerate optimizes the composition of skeletons at *run-time* rather than compile-time, and kernels are also compiled at run-time. For Scala, the Delite [16] framework can be considered as a skeletal parallelism approach. The goal of this framework is to ease the development of very high-level domain specific languages. All these languages target the Delite framework that is 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. PySke does not target GPUs yet. But it can run on both shared and distributed memory architectures and its set of skeletons is larger than the mentioned approached. Moreover it supports parallel trees.

Several skeleton libraries exist also for mainstream host sequential programming languages, for example for C++ [17], [18], C [19] or Java [20]–[22]. Compared to these libraries, PySke provides the same set of core classical skeletons and some original skeletons such as `get_partition` and `flatten`. PySke provides only data-parallel skeletons. The set of skeletons we provide in PySke is a super-set of a sub-set of the OSL [5] library for C++. Compared to OSL, PySke lacks a skeleton [23] to manage exceptions in parallel. OSL also provides `bh` a parallel parallel skeleton well-suited for bulk synchronous parallelism [24]. PySke does not provide this skeleton yet. OSL is close of the SkeTo library for C++, but SkeTo [2] also provides skeletons for multi-dimensional arrays. The current version of SkeTo does not provide tree skeletons but a previous one did [25]. Recent work considers a new implementation [26] that is not yet included in the current version of SkeTo.

III. Algorithmic Skeletons on Lists

The PySke algorithmic skeletons on lists are provided as methods of a class `PList` (for parallel list). PySke provides a global view of programs, i.e. a PySke program on parallel lists is written as a sequential program on sequential lists (PySke also offers a class `SList` with additional sequential functions on lists), but operates on parallel lists. This is very different from the programming style of MPI and mpi4py. Both follow the Single Program Multiple Data paradigm (SPMD) where the overall program must be thought as a parallel composition of sequential communicating programs.

Figure 2 illustrates the difficulty to read SPMD programs. The overall parallel program is the parallel composition of this program where the variable `pid` is giving the process identifier. While in sequential, the two branches of the conditional cannot be both executed, in SPMD they are both executed (if the number of processors is more than one). In this program processor 0 sends its pid to all other processors that in turn receive a value from processor 0 and then print their pids and the received value. The explanation of this program shows that it would be better to have the code performed by 0 before the code performed by the other processors. In this case the code could be changed to satisfy this constraint, but it is not always the case. When a program is complex it is also difficult to know if the value of a variable depends on the pid or not. Finally it is also difficult to determine if a variable is supposed to be used as a local sequential variable (like the loop counter `i`), or if the variable could be understood as a kind of array of size the number of processors.

The global view of PySke avoids these difficulties and makes the overall structure of parallel programs clearer.

```
from mpi4py import MPI
pid, nprocs = MPI.COMM_WORLD.Get_rank(),
    MPI.COMM_WORLD.Get_size()
if pid!=0:
    x = MPI.COMM_WORLD.recv(source=0);
    print("pid=", pid, "\nx=", x)
else:
    for i in range(1,nprocs):
    MPI.COMM_WORLD.send(pid, dest = i)
```

Fig. 2: A mpi4py SPMD Program

In addition to the default constructor that returns an empty parallel list, our API provides several ways to create a `PList`:

- the `init(f, size)` factory builds a parallel list of global size `size` that contains value `f(i)` at index `i`. Internally each processor contains a list of size `size / nprocs` where `nprocs` is the number of processors running the PySke program. If `size` is not dividable by `nprocs`, the first `size % nprocs` processors contain one more element than the other processors.
- the `from_seq(l)` factory builds a parallel list that contains only `l` at processor 0. The distribution of this list is not even. `l` does not need to be defined on processors

Global View	SPMD View			
	processor	0	1	2	3
$[0,1,2,3,4,5,6,7,8,9]$	content	$[0,1,2]$	$[3,4,5]$	$[6,7]$	$[8,9]$
	global_size	10	10	10	10
	local_size	3	3	2	2
	start_index	0	3	6	8
	distribution	$[3,3,2,2]$	$[3,3,2,2]$	$[3,3,2,2]$	$[3,3,2,2]$

Fig. 1: Global and SPMD view of `PList.init(lambda x:x,10)`

other than 0. This kind of factory can be useful when data can only be read from processor 0.

Internally, the implementation follows the SPMD style. A parallel list contains the following fields *on each processor*:

- `__distribution` is a list of numbers: it contains the local sizes for all the local contents. Therefore `__distribution` has length `nprocs`,
- `__content` contains the local piece, at a given processor, of the global list; the content of this field may be different on different processors,
- `__local_size` contains the size of `__content`, and for a processor with processor identifier `pid`, `__distribution[pid]` equals `__local_size`,
- `__global_size` contains the global size of the parallel list, i.e. the sum of all `__local_size`; the value is the same on all the processors,
- `__start_index` is the index in the global list of the first element of the local list.

Figure 1 shows a global view and its corresponding SPMD implementation of a parallel list build using `PList.init(lambda x:x,10)` on a machine with 4 processors.

The API then provides methods to apply a given sequential function to all the elements of one or two parallel lists, yielding a new parallel list. There are several variants of this `map` skeleton:

- For a parallel list `pl` and a unary function `f`, `pl.map(f)` is the parallel list obtained by applying `f` to each element of `pl`. This skeleton does not require any communication to be executed and the distribution of the output parallel list is the same as `pl`.
- For a parallel list `pl` and binary function `f` taking an index and a value, `pl.mapi(f)` is the parallel list obtained by applying `f` to each (global) index and the element at this index.
- For two parallel lists `pl1` and `pl2`, and a binary function `f`, `pl1.map2(f,pl2)` is the parallel list obtained by applying `f` at every possible index to the element of `pl1` and the element of `pl2`. A pre-condition for this skeleton to execute correctly is that `pl1` and `pl2` have the same distribution (and hence the same size).
- The `zip` skeleton is just a call to `map2` where `f` builds a pair from two values.

The first skeleton that needs communications for its execution is the `reduce` skeleton. Using a binary operation `op`, which forms a monoid with value `e` (i.e. `op` is associative and for all value `x`, `op(x,e)` equals `op(e,x)` equals `x`), `pl.reduce(op, e)` returns the value `op(pl[__global_size-1,op(...,op(pl[0],e))`. If `pl` is non empty, then `e` can be omitted. For example the sum of all the elements of a parallel list `pl` can be written `pl.reduce(lambda x,y:x+y)`. The result of `reduce` is a sequential value.

Since the computation patterns are both defined for `SList` and `PList` instances, the following program computes the variance of a discrete random variable `X` representing either a sequential list, or a parallel list:

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

There is no difference between the sequential program and its parallel version. This transparency aims at keeping the same productivity for programmers.

The `map` skeleton and variants cannot change the distribution of their input parallel lists. However it may be necessary to do so, for example to filter out some of the elements of the distributed list. In order to obtain a flexible mechanism, we provide a skeleton named `get_partition` that is more general than a `filter` skeleton. `get_partition` basically changes the view that the users have of the parallel list. Instead of being a list of elements, `pl.get_partition()` is a parallel list of `nprocs` *lists*. The distribution of `pl.get_partition()` is the list of size `nprocs` containing only `1`. On 4 processors, if the global view of `pl` was the one of Figure 1, then the global view of `pl.get_partition()` is `[[0,1,2],[3,4,5],[6,7],[8,9]]`. Now a simple application of `map` is enough to filter out some values, for examples all the values below 5:

```
pl2 = pl.get_partition().map(lambda
    l:l.filter(lambda x: x>5))
```

The global view of the resulting list is: `[[],[],[6,7],[8,9]]`.

The skeleton `flatten` allows to obtain a list of elements from a parallel list of lists. `pl3 = pl2.flatten()` has the global view `[6,7,8,9]`, but it is not evenly distributed. Note that to have a consistent distribution information on all processors, the local sizes should be broadcast. The distribu-

Global View

$[[a],[b,d,e],[c,f,g],[h,j,k],[i,l,m]]$

SPMD View

processor	0	1	2	3
content	$[a,b,d,e]$	$[c,f,g]$	$[h,j,k]$	$[i,l,m]$
distribution	[2,1,1,1]	[2,1,1,1]	[2,1,1,1]	[2,1,1,1]
global_index	[(0,1),(1,3),(0,3),(0,3),(0,3)]			
start_index	0	2	3	4
nb_segs	2	1	1	1

Fig. 3: Global and SPMD view of `PTree(lt)` (`lt` from Figure 4)

tion is this case is `[0,0,2,2]`.

The skeleton `balance` returns a parallel list that is globally equivalent to the input object, but that is evenly distributed. Thus the distribution of `pl3.balance()` is `[1,1,1,1]`.

IV. Algorithmic Skeletons on Trees

Trees are a particular kind of graphs, often used to represent structured data such as organizational charts or XML documents. However, their ill-balanced and irregular structures make efficient parallel computations challenging. Contrary to lists, the structure of trees is not linear. Computations therefore proceed from top to bottom (or from bottom to top), instead of from left to right (or right to left).

A. Binary Trees

A binary tree, `BTree`, is a tree in which a node has two children. Two constructors are defined by inheritance. `Leaf(a)`, to instantiate a binary tree with only one element containing the value `a`, and `Node(b,lb,rb)` where `b` is the value contained into the node, with `lb` and `rb` two binary trees corresponding to the children nodes.

B. Serialization and Distribution

A list can be easily cut into contiguous pieces and these pieces distributed. The structure of a binary tree does not allow the same distribution strategy. However, a tree can be divided as follows: Given an integer `m`, a node `v` is called m-critical if, for each `v'` child of `v`, the following inequality is respected: $\lceil size(v)/m \rceil > \lceil size(v')/m \rceil$ with

```
size = lambda v: 1 + (0 if v.is_leaf()
       else v.left.size() + v.right.size())
```

The critical nodes are the cut points of the tree. Each subtree is translated into a list of `TaggedValue` called `Segment` and encapsulated into an `LTree` instance. A `TaggedValue` is a couple of a value and a tag corresponding to the type of element in the original tree. Figure 4 gives an example of a serialized tree with $m = 5$. Tags are `L` for leaf, `C` for critical node, and `N` for regular node.

The PySke algorithmic skeletons on linearized trees are provided as methods of a class `PTree` (for parallel tree). They are built with the same approach as parallel lists. The two ways to create a parallel tree are the following:

- The default constructor of `PTree` distributes a linearized tree, and can be called by `PTree(lt)`. The distribution is not based on the number of `Segment` but on the average number of `TaggedValue` in a global repartition. Obviously, depending on the serialization parameters, all the `Segment` won't have the same number of elements. The distribution is built to have a number of values close to `total_size / nprocs`. If used without any input, the constructor will return an empty parallel tree.
- A `PTree` can be imported from a text file, using the factory `PTree.init_from_file(filename, parse)` with `parse` a parser from string to the type of value contained in the tree (`int` by default).

The SPMD style is also followed for the *implementation* of parallel trees (but the users' API does follow the global view approach). The fields of a `PTree` are:

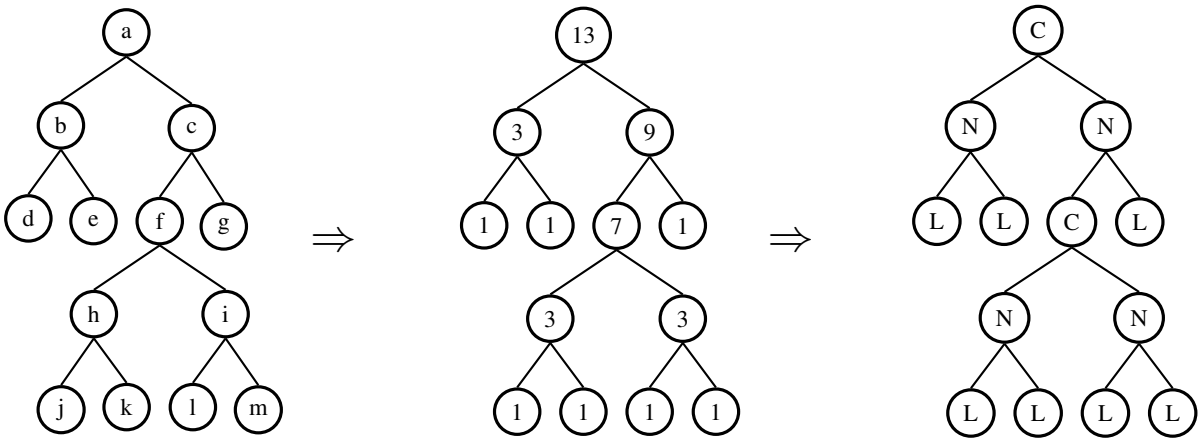
- `__content` contains a single list of `TaggedValue` representing all the `Segment` contained in the current instance.
- `__distribution` is similar than for parallel lists. It is a list of number representing the number of `Segment` per processor.
- `__global_index` contains all the index of the distribution of the segments. The index is a list of a couple of integers representing the start point of a segment and its size. The start points are calculated for each processor, i.e. the start point of the first segment of a processor is always 0.
- `__start_index` index of first index for the current `pid` in global_index
- `__nb_segs` contains the number of `Segment` in the local content. This value can be got using `__distribution[pid]`.

Figure 3 shows the global view and the corresponding SPMD implementation of a parallel tree build using `PTree(lt)` with `lt` the linearized tree presented in Figure 4, on a machine with 4 processors. To simplify the notation, we just represent the `TaggedValue` instances by their values.

C. Skeletons

The tree skeletons described by Skillicorn in [27] represent the basis of the tree skeletons implemented in the library. The patterns on trees follow the ones on lists.

The first one, the `map` skeleton, applies two functions to every element of a tree. One of the function is applied to leaf values, while the second one is applied to node values. For a parallel tree `pt`, `pt.map(kL, kN)` is the parallel tree



(a) tree (b) sizes (c) tags

```
s1 = Segment([TaggedValue(a,'C')])
s2 = Segment([TaggedValue(b,'N'), TaggedValue(d,'L'), TaggedValue(e,'L')])
s3 = Segment([TaggedValue(c,'N'), TaggedValue(f,'C'), TaggedValue(g,'L')])
s4 = Segment([TaggedValue(h,'N'), TaggedValue(j,'L'), TaggedValue(k,'L')])
s5 = Segment([TaggedValue(i,'N'), TaggedValue(l,'L'), TaggedValue(m,'L')])
lt = LTree([s1,s2,s3,s4,s5])
```

Fig. 4: An example of a list representation of a binary tree (with $m = 5$).

obtained by applying `kL` to each leaf value and `kN` to each node value of `pt`. This skeleton is pretty simple because it does not require any communication and then can be executed on a single step. The skeletons `zip` and `map2` are defined with the same approach. For two parallel tree `pt1`, and `pt2` with the exact same shape (same distribution, and same tags on values), `pt1.map2(pt2, f)` constructs a new `PTree` where the values are obtained by applying `f` at every possible index to the element of `pt1` and the element of `pt2`. The `zip` skeleton is defined as a particular case of `map2` where `f = lambda x, y: (x,y)`, and can be called by `pt1.zip(pt2)`.

The `reduce` skeleton is based on its sequential definition defined by:

```
Leaf(a).reduce(k) == a
Node(b,lb,rb).reduce(k) == k(reduce(k,lb), b,
                            reduce(k,rb))
```

However, because of the computation dependencies, k must allow partial calculation on subparts of a tree. The calculation of k can be partially defined if there exists ϕ, ψ_N, ψ_L, and ψ_R such that:

$$\begin{cases} k(l,b,r) & = & \psi_N(l,\phi(b),r) \\ \psi_N(\psi_N(x,l,y),b,r) & = & \psi_N(x,\psi_L(l,b,r),y) \\ \psi_N(l,b,\psi_N(x,r,y)) & = & \psi_N(x,\psi_R(l,b,r),y) \end{cases}$$

This closure property is written $k =< \phi, \psi_N, \psi_L, \psi_R >_u$. The `reduce` skeleton can be then used on a parallel tree `pt` by `pt.reduce(k, phi, psiN, psiL, psiR)`. Its execution necessitates communications and is composed of several steps: First, each `Segment` is locally reduced into a single value with `k`, `phi`, `psiL` and `psiR`. After all the local results are gathered at processor 0, a global reduction is computed using `k` and `psiN`. The `reduce` skeleton returns either a single value in the first processor and `None` otherwise.

The Upward Accumulation function, `uacc`, is defined similarly but it has the particularity of keeping the tree structure for its result:

```
Leaf(a).uacc(k) == Leaf(a)
Node(b,lb,rb).uacc(k) == Node(k(lb.reduce(k),
    b, rb.reduce(k)), lb.uacc(k), rb.uacc(k))
```

In the same way, to allow parallelization, the closure property $k =< \phi, \psi_N, \psi_L, \psi_R >_u$ must be respected.

The `uacc` function is used as follows:

```
pt.uacc(k, phi, psiN, psiL, psiR).
```

Three computation steps are necessary for the parallel execution of `uacc`. We first make a local accumulation processed with `k`, `phi`, `psiL` and `psiR` to get both local, but incomplete, accumulated segments, and the top values of accumulations (later used to process complete accumulation). The calculated top values are gathered to the processor such that `pid == 0`, and with `psiN`, the actual top values are calculated. The actual top values are redistributed and each `Segment` is finally updated if necessary during one last step, using `k`.

The definition of the Downward Accumulation function, `dacc`, is the following:

```
Leaf(a).dacc(gL, gR, c) == Leaf(c)
Node(b, lb, rb).dacc(gL, gR, c) ==
  Node(c,lb.dacc(gL, gR, gL(c,b)),
         rb.dacc(gL, gR, gR(c,b)))
```

Here again, the used functions must respect a closure property. The `dacc` function can be parallelized if there exists phi_L, phi_R, psi_U and psi_D such that:

$$\begin{cases} g_L(c,b) & = & \psi_D(c,\phi_L(b)) \\ g_R(c,b) & = & \psi_D(c,\phi_R(b)) \\ \psi_D(\psi_D(c,b),b') & = & \psi_D(c,\psi_U(b,b')) \end{cases}$$

The closure property on (g_L, g_R) is denoted by:

$$(g_L, g_R) =< \phi_L, \phi_R, \psi_U, \psi_D >_d .$$

`pt.dacc(gL, gR, c, phiL, phiR, psiU, psiD)` is therefore a call to the `dacc` skeleton.

This skeleton is also processed using several computation steps. First, each processor computes a local intermediate value with `psiU`, `phiL` and `phiR`. Theses values are gathered at processor 0. Using `psiD` and the initial value of `c`, it computes values to pass to children of critical nodes. After having being redistributed, these values are used by each processor to perform a global downward accumulation.

Numbering the nodes following a prefix traversal order of a tree `T`, implemented as a parallel tree, can be computed by:

```
def k((ll,ls), b, (rl,rs)):
  return (ls, ls + 1 + rs)
initial = T.map(lambda a: (0,1),lambda x: x)
processed = initial.uacc(k, phi, psiN, psiL,
    psiR)
prefixed = processed.dacc(gL, gR, 0, phiL,
    phiR, psiU, psiD)
```

All the source code is available at https://pypi.org/project/pyske/, or can be installed by the following command:

```
# pip install pyske
```

Auxiliary functions used for closure properties are obtained using the derivation technique described in [28]. The skeletons described above have their sequential implementations for `LTree`. For an instance of distributed tree `pt`, if there exists a skeleton `F`, then there exists the same function that can be called by `lt.F(params)` with `lt` an instance of `LTree`. The functions can also be called on `BTree` instances, but all the input parameters relative to closure properties must be removed. The sum of nodes can then be processed as following, with `lt` an instance of `LTree` and `bt` an instance of `BTree`.

```
def add(x,y,z): return x + y + z
def one(x): return 1
sum_lt = lt.reduce(add, one, add, add, add)
sum_bt = bt.reduce(add)
```

V. Examples and Experiments

We present here two experiments we processed using PySke. Each example ran 30 times. The execution time reported is the average over the 30 experiments of the maximum value of the execution times of all the MPI processes. Note that unlike the default of the `timeit` Python library, we do not exclude garbage collection of our timings.

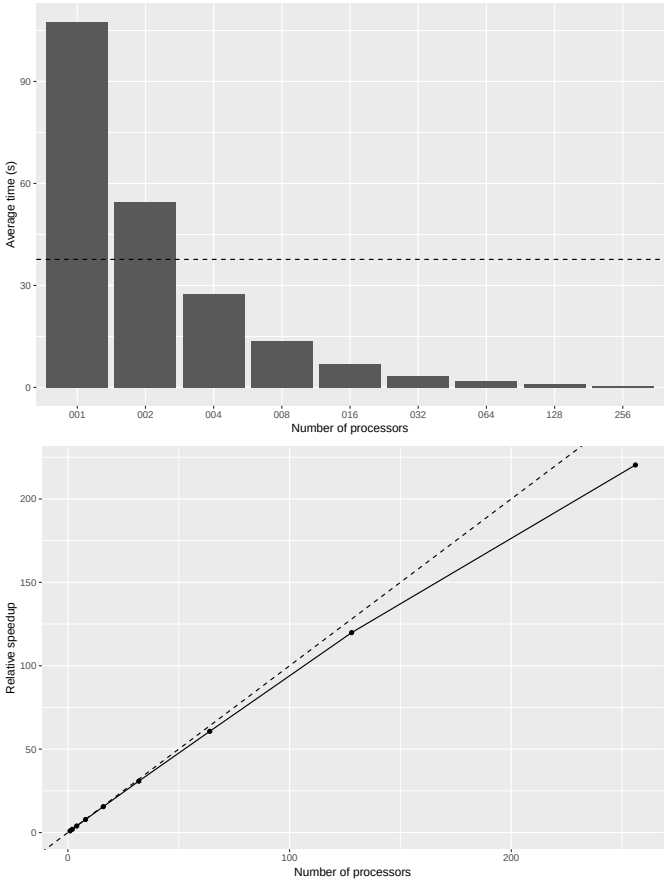


Fig. 5: Example: Variance (List)

The experiments have been processed on Monsoon, the HPC cluster of Northern Arizona University. The nodes of the cluster have 16 Intel Xeon cores, with a total of 24TB of memory. Individual systems are interconnected via FDR Infiniband at a rate of 56Gbps. The used software is the following: Ubuntu Linux 18.04, Python 3.6.7, mpi4py version 3.0.0, OpenMPI version 2.1.1. The choice of using a cluster has been taken to mainly test the scalabilty of PySke programs. The same executions on a single shared memory machine, that has two Intel Xeon E5-2683 v4 processors with 16 cores at 2.10 GHz, and 256Gb of memory, took half the time. Since this second option has a limited number of cores (i.e., 32), we preferred show our results on Monsoon.

The first one is the application of the variance calculation on a distributed list of $5 * 10^7$ integer elements. Figure 5 presents the calculation time and the relative speed-up depending on the number of processors. The dashed line in the first plot represents the computation time using just plain Python for the same problem, while the one in the second graph represents a perfect speed-up.

Figure 6 presents results of experiments on the node numbering application on trees. We have performed the tests on two types of binary tree of size $2^{24}-1 = 16777215$: a balanced tree, and a tree with a random shape. All the trees have been

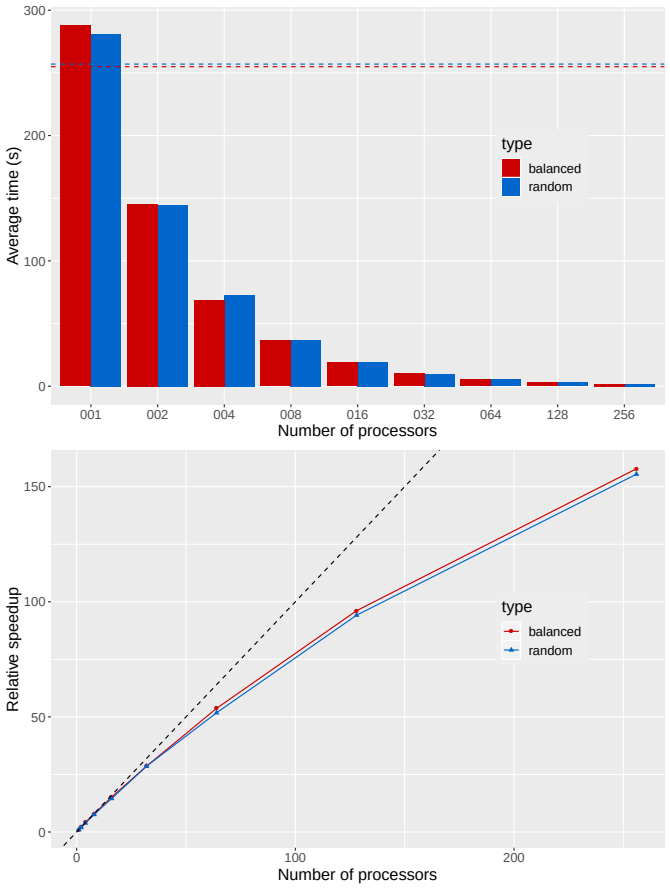


Fig. 6: Example: Node Numbering (Tree)

linearized using $m = 53600$. The dashed lines represent the computation times using `LTree` instead of `PTree` instances. Compared to `BTree`, linearized trees are much more efficient. The recursive functions on binary trees are not tail recursive: that makes their computations inefficient. In average, it took 697s (resp. 745s) to compute `prefix` on a balanced binary tree (resp. random shaped binary tree) with `BTree` primitives.

The results of the parallel implementation using PySke skeletons show a good scalability until 128 processors. The results are better for balanced tree because of the more predictable distribution of the tree. However, even with 256 processing units, the performances are increasing. Python implies an evident performance penalty. For example, compared to SkeTo, the same program is slower but the relative speed-up increases similarly. Figure 7 shows a comparison of the relative speed-up on same executions of the node numbering program. We do not have a comparison of the execution time because the values for SkeTo come from a paper that was reporting performances for a very different machine.

The value of m has a real importance in the distribution of the data. If m has a small value, the linearized `Segment` of tree will be small too. It is convenient to balance the distribution within the processors but it implies to do more partial calculation. A large value makes less `Segment` but the

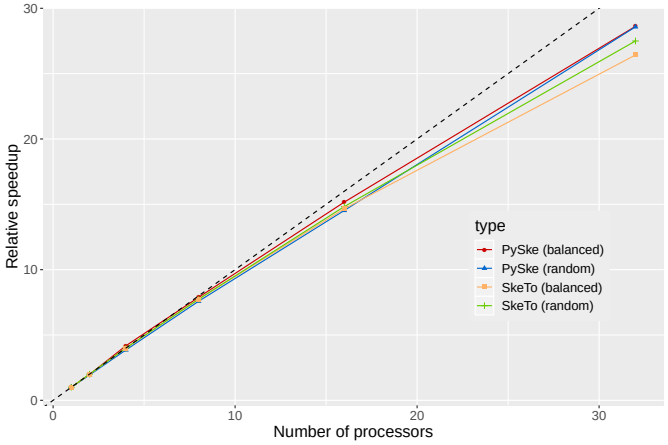


Fig. 7: Comparison of Relative-Speed (Tree)

distribution may be more unbalanced. Matsuzaki et. al. discuss more precisely about this value in [25]. The `map` skeleton by itself would show a better relative speed-up because of the absence of communications. With a perfect distribution ($m = 1$), the scalability would be perfect. The choice of how splitting the trees by considering the number of processors, the machine characteristics, and the size of each subtree with more relativity would be better. The technique of distribution can be improved by detecting the type of a tree. For a given node, by calculating the size of the left and the right subtrees, the type of the binary tree can be decided, and then the decision of splitting.

VI. Conclusion and Future Work

In this paper, we presented PySke, a Python library of skeletons for lists and binary trees. Skeletal parallelism eases writing parallel programs compared to more explicit approachs. We proposed different applications based on PySke skeletons, and showed the scalability of our solution.

The API can be completed with other skeletons. First, the already defined structures can provide more parallel patterns especially on lists [2]–[5]. Also, other structures with their skeletons such as matrices [29], graphs [30], or more general trees [31] will be implemented in future versions of PySke.

Providing more structures and more skeletons will increase productivity but it can also decrease the performances if the skeleton combinations are not optimized. Program optimizations can be automatically performed using program transformations [32]–[35]. The program process can be guided by a cost model [36]. Preliminary results show that it is indeed a promising direction for PySke [37].

On the implementation side, the *mpi4py* library has been particularly designed and optimized for numeric arrays of the *NumPy* [38] library. We plan to provide a distributed array data structure based on contiguous *NumPy* arrays, and to provide the same skeletons than for lists.

The *mpi4py* library is not the only one that allows parallel programming in Python. We plan to comparatively study

the different Python programming libraries, both from the runtime performance perspective but also from the productivity perspective. Halstead metrics [39] and similar metrics are well-designed to evaluate the effort needed to write the same program in different ways. It already have been used by Légaux et al. [40], and we plan to use such metrics to provide a comparison of PySke with other parallel programming libraries.

REFERENCES

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

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

[3] 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.

[4] 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.

[5] 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.

[6] L. Dalcin, R. Paz, and M. Storti, "MPI for Python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108 – 1115, 2005.

[7] L. Dalcin, R. Paz, M. Storti, and J. D'Elia, "MPI for Python: Performance improvements and MPI-2 extensions," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 655 – 662, 2008.

[8] 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.

[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, F. Gava, and D. Billiet, "Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction," in *International Conference on Computational Science (ICCS)*, ser. LNCS, vol. 3515. Springer, 2005, pp. 1046–1054.

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

[13] R. Loogen, Y. Ortega-Mallen, and R. Pena-Mari, "Parallel Functional Programming in Eden," *J Funct Program*, vol. 3, no. 15, pp. 431–475, 2005.

[14] R. Loogen, "Eden – Parallel Functional Programming with Haskell," in *Central European Functional Programming School*, ser. LNCS, V. Zsòk, Z. Horváth, and R. Plasmeijer, Eds. Springer, 2012, vol. 7241, pp. 142–206.

[15] 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.

[16] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. Embed. Comput. Syst.*, vol. 13, pp. 134:1–134:25, Apr. 2014.

[17] J. Falcou, J. Sérot, T. Chateau, and J.-T. Lapresté, "Quaff: Efficient C++ Design for Parallel Skeletons," *Parallel Computing*, vol. 32, pp. 604–615, 2006.

[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] A. Benoit, M. Cole, S. Gilmore, and J. Hillston, "Flexible Skeletal Programming with eSkel," in *11th International Euro-Par Conference*, ser. LNCS 3648, J. C. Cunha and P. D. Medeiros, Eds. Springer, 2005, pp. 761–770.

[20] M. Danelutto and P. Dazzi, "Joint Structured/Unstructured Parallelism Exploitation in Muskel," in *International Conference on Computational Science (ICCS)*, ser. LNCS. Springer, 2006.

[21] D. Caromel and M. Leyton, "Fine tuning algorithmic skeletons," in *Euro-Par Parallel Processing*, ser. LNCS 4641, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds., vol. 4641. Springer, 2007, pp. 72–81.

[22] M. Leyton and J. M. Piquer, "Skandium: Multi-core Programming with Algorithmic Skeletons," in *PDP*. IEEE, 2010, pp. 289–296.

[23] 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.

[24] J. Légaux, Z. Hu, F. Loulergue, K. Matsuzaki, and J. Tesson, "Programming with BSP Homomorphisms," in *Euro-Par Parallel Processing*, ser. LNCS, no. 8097. Aachen, Germany: Springer, 2013, pp. 446–457.

[25] K. Matsuzaki, "Efficient Implementation of Tree Accumulations on Distributed-Memroy Parallel Computers," in *International Conference on Computational Science (ICCS)*, ser. LNCS, vol. 4488. Springer, 2007.

[26] S. Sato and K. Matsuzaki, "A Generic Implementation of Tree Skeletons," *Int J Parallel Prog*, 2015.

[27] D. Skillicorn, "Parallel implementation of tree skeletons," *J. Parallel Distrib. Comput.*, vol. 39, no. 2, pp. 115–125, Dec. 1996.

[28] K. Matsuzaki, Z. Hu, and M. Takeichi, "Towards automatic parallelization of tree reductions in dynamic programming," in *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '06. New York, NY, USA: ACM, 2006, pp. 39–48.

[29] K. Emoto, Z. Hu, K. Kakehi, and M. Takeichi, "A compositional framework for developing parallel programs on two-dimensional arrays," *Int. J. Parallel Program.*, vol. 35, no. 6, pp. 615–658, Dec. 2007.

[30] K. Emoto, K. Matsuzaki, Z. Hu, A. Morihata, and H. Iwasaki, "Think like a vertex, behave like a function. a functional DSL for vertex-centric big graph processing," in *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2016, pp. 200–213.

[31] K. Matsuzaki, Z. Hu, and M. Takeichi, "Parallel skeletons for manipulating general trees," *Parallel Computing*, vol. 32, no. 7-8, pp. 590–603, 2006.

[32] J. Gibbons, "The third homomorphism theorem," *J Funct Program*, vol. 6, no. 4, pp. 657–665, 1996.

[33] A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi, "The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer," in *POPL'09*, Z. Shao and B. C. Pierce, Eds. ACM, 2009, pp. 177–185.

[34] A. Rasch and S. Gorlatch, "Multi-dimensional homomorphisms and their implementation in OpenCL," *International Journal of Parallel Programming*, vol. 46, no. 1, pp. 101–119, Feb 2018.

[35] 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)*. ACM, January 22-23 1999, pp. 85–94.

[36] M. Aldinucci, M. Danelutto, J. Dünnweber, and S. Gorlatch, "Optimization techniques for skeletons on grids," in *Grid Computing The New Frontier of High Performance Computing*, ser. Advances in Parallel Computing, L. Grandinetti, Ed. North-Holland, 2005, vol. 14, pp. 255 – 273.

[37] J. Philippe and F. Loulergue, "Towards automatically optimizing PySke programs (poster)," in *International Conference on High Performance Computing and Simulation (HPCS)*. Dublin, Ireland: IEEE, 2019.

[38] Oliphant T., "NumPy: numerical Python," http://numpy.scipy.org, 2010.

[39] T. Hariprasad, G. Vidhyagaran, K. Seenu, and C. Thirumalai, "Software complexity analysis using halstead metrics," in *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, May 2017, pp. 1109–1113.

[40] 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.