PySke: Algorithmic Skeletons for Python

Jolan Philippe[1,2] and Frédéric Loulergue[1,3]

[1]Northern Arizona University - Flagstaff, AZ (USA)
[2]IMT Atlantique - Nantes (France)
[3]Université d'Orléans, LIFO - Orléans (France)

October 31, 2019

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement n° 813884.

Write a parallel program is a difficult task for casual programers.
Example: duplicated code on each processor for average calculation (Python):

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
pid, nprocs = comm.Get_rank(), comm.Get_size()

def average(data):
  size = len(data)

  min = 0
  for i in range(pid):
    min += int(size / nprocs) + (1 if i < size % nprocs
    else 0)
  max = min + int(size / nprocs) + (1 if pid < size %
    nprocs else 0) + 1

  local_sum = sum(data[min:max])
  global_sum = comm.allreduce(local_sum, op=MPI_SUM)
  return global_sum / size
```

Python + Skeletons

Difficulties:

- Communications must be explicit: whom? what?
- Distribution: how make a balanced distribution
- Error-prone: low-level primitives use
- ...

Python + Skeletons

Difficulties:

- Communications must be explicit: whom? what?
- Distribution: how make a balanced distribution
- Error-prone: low-level primitives use
- ...

Our solution: PySke , a Python Skeletons library

PySke

PySke aims at easing parallel programming on data-structures.

- On top of `mpi4py` (MPI library for Python)
- Distributed data
- Same program executed by each processor (SPMD)
- A global view for the users

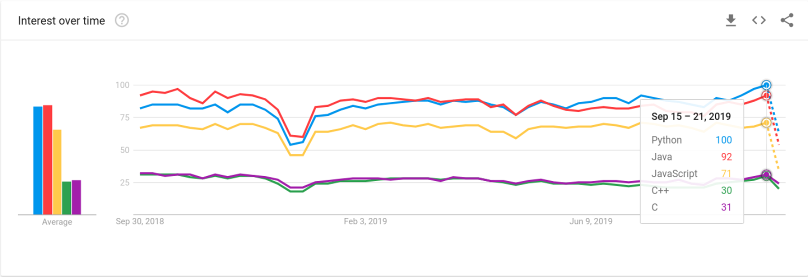
Python

Why Python?

- Python is cool (but pythons are not)
- OOP and functional programming aspects (e.g., lambda, high-order functions)
- A popular language in the programming community
- Academic-friendly for informatics (applied CS)

Python

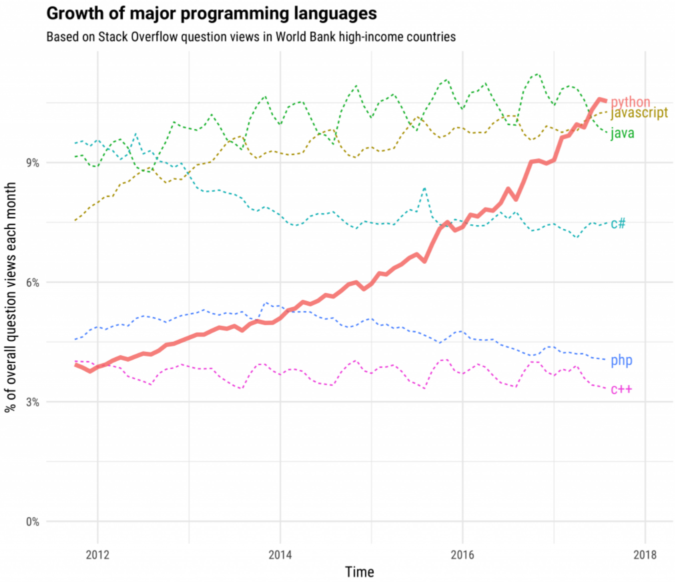
One of the most searched language on Google



Source: Google Trends

Python

Stack Overflow questions



Source: StackOverflow blog

Skeletons

Wikipedia: "In computing, algorithmic skeletons, or parallelism patterns, are a high-level parallel programming model for parallel and distributed computing."

- Parallel implementation of a computation pattern
- A high abstraction for parallelism
- Defined by Murray Cole (1989)

PySke targets skeletons on distributed data-structures.

Skeletons libraries

SkeTo	C++	Multidimensional arrays, lists, matrices
SkePu	C++ (GPU)	Arrays, vectors
Accelerate	Haskell (GPU)	Array
Muskel	Java/RMI	Clusters, networks, and grids
OSL	C++	Lists and exceptions
Delite	C++ (CPU and GPU)	Compiler
parmap	OCaml	Lists
BSML	OCaml	Vectors
DatTel	C++	Templates
Muesli	C++ (CPU and GPU)	Arrays, (sparse) matrices, tasks
SkelGIS	C++	2D Cartesian meshes (Scientific Simulations)
eSkel	C	Tasks
MaLLBa	C++	Tasks
OCamlP3L (and Skml)	OCaml	Tasks
Lithium, Calcium, Skandium	Java	Tasks
Eden	Haskell	Process
Quaff	C++	Tasks

MapReduce, Hadoop, Pregel, Spark, etc. can be considered as skeletal architectures

Skeletons libraries

SkeTo	C++	Multidimensional arrays, lists, matrices
SkePu	C++ (GPU)	Arrays, vectors
Accelerate	Haskell (GPU)	Array
Muskel	Java/RMI	Clusters, networks, and grids
OSL	C++	Lists and exceptions
Delite	C++ (CPU and GPU)	Compiler
parmap	OCaml	Lists
BSML	OCaml	Vectors
DatTel	C++	Templates
Muesli	C++ (CPU and GPU)	Arrays, (sparse) matrices, tasks
SkelGIS	C++	2D Cartesian meshes (Scientific Simulations)
eSkel	C	Tasks
MaLLBa	C++	Tasks
OCamlP3L (and Skml)	OCaml	Tasks
Lithium, Calcium, Skandium	Java	Tasks
Eden	Haskell	Process
Quaff	C++	Tasks

MapReduce, Hadoop, Pregel, Spark, etc. can be considered as skeletal architectures

$\Rightarrow$ Lack of skeletons on trees. + No library in Python

An example

Variance formula: $V = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^2$ with $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$

With PySke (global view):

```
(* add = lambda x, y: x + y *)
def variance(l: List[float]) -> float:
  n = l.length()
  xbar = l.reduce(add) / n
  v = l.map(lambda num: (num-xbar)**2).reduce(add) / n
  return v
```

An example

Variance formula: $V = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^2$ with $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$

With PySke (global view):

```
(* add = lambda x, y: x + y *)
def variance(l: List[float]) -> float:
  n = l.length()
  xbar = l.reduce(add) / n
  v = l.map(lambda num: (num-xbar)**2).reduce(add) / n
  return v
```

Why `List` and not `list`? An interface for lists in PySke (will be more detailed later)

Global view

```
def variance(l: List[float]) -> float:
  n = l.length()
  xbar = l.reduce(add) / n
  v = l.map(lambda num: (num-xbar)**2).reduce(add) / n
  return v
```

Old difficulties:

- Communications: implicit communications
- Distribution: already distributed structures
- Error-prone: use of defined skeletons

New problematic:

- Composition: How to write a program using PySke?

Types in PySke

Two structures:

1. Lists
   - `SList`, an extension of `list`, with OOP style
   - `PList`, distributed lists
2. Trees
   - `BTree`, asbtract class for binary trees, extended by `Node` and `Leaf`
   - `LTree`, linearized trees
   - `PTree`, distributed trees
   - (`RNode`, rose trees (arbitrary shape), but only sequential)

Primitives on lists

Instanciations:

- `SList()` and `PList()` for empty lists
- `SList([x,y,z])`: instantiate a list containing `x`, `y` and `z`
- `PList.init(f, size)`: instantiate a distributed list of length `size` and at the index `i`, `f(i)`
- `PList.from_seq(l)`: instantiate a distributed list from a sequential one

`PList` also contains a method `to_seq` to get a `SList` from a distributed list

Primitives on lists

Skeletons, same signature in `SList` and `PList` classes:

- `map(f)` and variants: `zip(l)`, `map2(op, l)`, `mapi(f)`, `map2i(op, l)`
- `reduce(op)` (op must be associative for parallelism)
- variants of scan: `scanr(op)`, `scanl_last(op, e)`, `scanl(op, e)`, `scanp(op, e)`
- `filter(p)`

Only for `PList`:

- `get_partition()`, `flatten()`, `distribute(l)`, `balance()`
- `gather(pid)`, `scatter(pid)` and `scatter_range(rng)`
- `permute(f)`

Implementation: PList

Global View	SPMD View				
	processor	0	1	2	3
	content	[0, 1, 2]	[3, 4, 5]	[6, 7]	[8, 9]
[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]

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

Implementation: Skeletons

An example: *map*

- independent sub-computation
- no communication

	Global view:	$map(f, [x_1, ..., x_i, x_{i+1}, ..., x_j, x_{j+1}, ..., x_n])$		
	Processors	p_1	p_2	p_3
0	Content	$[x_1, ..., x_i]$	$[x_{i+1}, ..., x_j]$	$[x_{j+1}, ..., x_n]$
1	Local *map*	$map(f, [x_1, ..., x_i])$	$map(f, [x_{i+1}, ..., x_j])$	$map(f, [x_{j+1}, ..., x_n])$
2	New content	$[f(x_1), ..., f(x_i)]$	$[f(x_{i+1}), ..., f(x_j)]$	$[f(x_{j+1}), ..., f(x_n)]$
	Global view:	$[y_1, ..., y_i, y_{i+1}, ..., y_j, y_{j+1}, ..., y_n]$ with $\forall i \in [1..n], y_i = f(x_i)$		

Implementation: Skeletons

Harder example: *scan*

- dependent computations
- two phases
- non-trivial communications

Python code snippet: parallelscan.py

⇒ The use of skeletons largely eases a program needing an accumulation of values

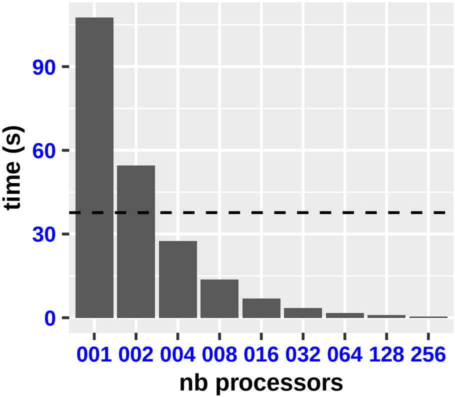
Example: Variance

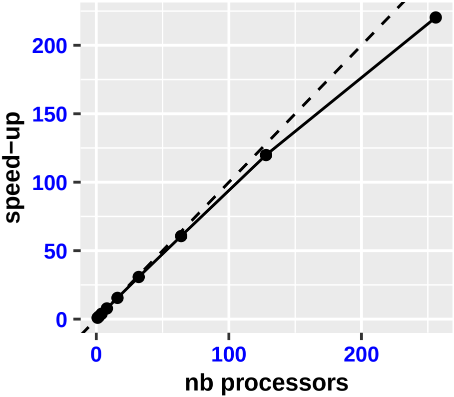
```
def variance(input: PList[float]) -> float:
...
```

For a parallel implementation, need to use the following skeletons: `map`, and `reduce`.

Example: Variance

Variance on a list of 5.10^7 integers





HPC cluster (total of 24TB of memory), 16 Intel Xeon cores per node. Individual systems are interconnected via FDR Infiniband at a rate of 56Gbps. Ran 30 times with the following software: Ubuntu Linux 18.04, Python 3.6.7, mpi4py 3.0.0, OpenMPI version 2.1.1.

Complex example: Fast Fourrier Transformation

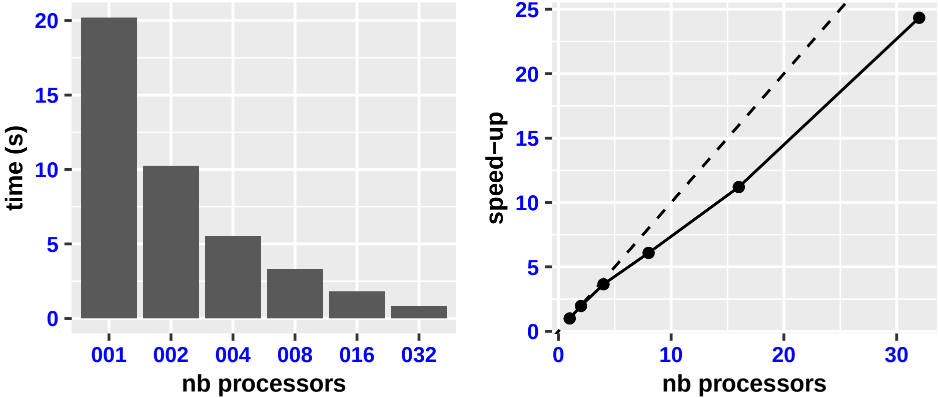
Wikipedia: Convert a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa

```
def fft(input: PList[float]) -> PList[complex]:
...
```

For a parallel implementation, need to use the following skeletons: `map`, `get_partition`, `permute`, `flatten`, and `map2i`.

Example: Fast Fourrier Transformation

Fast Fourrier Transformation on a list of 2^{18} floating point numbers



Shared memory machine (256 Gb), two Intel Xeon E5-2683 v4 (16 cores at 2.10 GHz).

Ran 30 times with the following software: CentOS 7, Python 3.6.3, mpi4py 3.0.2, OpenMPI 2.6.4.

Primitives on trees

Instanciations:

- `Leaf(v)` and `Node(v, l, r)` for binary trees
- `LTree` extends `SList`, adding `LTree.init_from_bt(bt, m)`
- `PTree(lt)`: distribute a linearized tree
- `PTree.init(pt, content)`: instantiate a new distributed tree with a new content

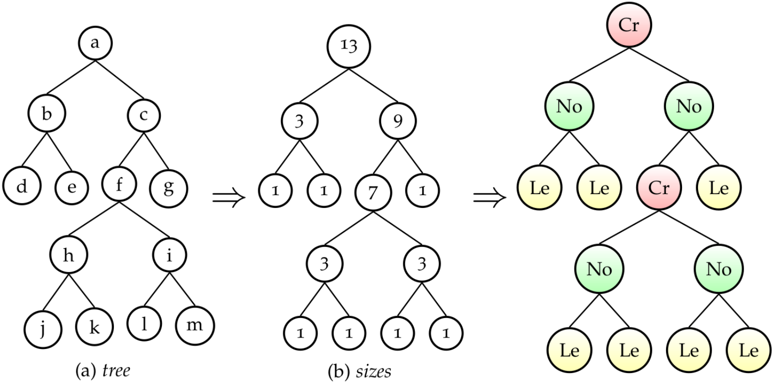
`PTree` also contains a method `to_seq` to get a `LTree` from a distributed tree

Serialization of a BTree into a LTree

For a value m, a node is critical if

- $\lceil size(Node(v, l, r))/m \rceil > \lceil size(l)/m \rceil$
- $\lceil size(Node(v, l, r))/m \rceil > \lceil size(r)/m \rceil$

Example with $m = 5$:



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

Primitives on trees

Skeletons, pink parametrers are only for `LTree` and `PTree` instances

- `map(fl, fn)` and variants: `zip(pt)`, `map2(op, pt)`
- `reduce(k, phi, psi_n, psi_l, psi_r)` (`k` must respect a closure property for parallelism)
- `uacc(k, phi, psi_n, psi_l, psi_r)` (`k` must respect a closure property for parallelism)
- `dacc(gl, gr, c, phi_l, phi_r, psi_u, psi_d)` (`gl` and `gr` must respect a closure property for parallelism)

The closure properties are based on Kiminori Matsuzaki et. al. works.

Closure property for reduce and uacc

Additional arguments for `reduce` and `uacc` respecting:

$$k : (A * B * A) \to A \qquad \psi_r : (A * C * C) \to C$$
$$\psi_n : (A * C * A) \to A \qquad \psi_l : (C * C * A) \to C$$
$$\phi : B \to C$$

$$\begin{aligned}
k(l, b, r) &= \psi_n(l, \phi(b), r) \\
\psi_n(\psi_n(x, l, y), b, r) &= \psi_n(x, \psi_l(l, b, r), y) \\
\psi_n(l, b, \psi_n(x, r, y)) &= \psi_n(x, \psi_r(l, b, r), y)
\end{aligned}$$

Closure property for dacc

Additional arguments for `dacc` respecting:

$$gl : (C * B) \to C \qquad gr : (C * B) \to C$$

$$\phi_l : B \to D \qquad \phi_r : B \to D$$

$$\psi_u : (C * D) \to D \qquad \psi_d : (C * D) \to C$$

$$\begin{array}{rcl} g_l(c, b) & = & \psi_d(c, \phi_l(b)) \\ g_r(c, b) & = & \psi_d(c, \phi_r(b)) \\ \psi_d(\psi_d(c, b), b') & = & \psi_d(c, \psi_u(b, b')) \end{array}$$

Implementation of PTree

Global View	SPMD View				
	processor	0	1	2	3
	content	$[a, b, d, e]$	$[c, f, g]$	$[h, j, k]$	$[i, l, m]$
$[[a], [b, d, e], [c, f, g], [h, j, k], [i, l, m]]$	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

Figure: Global and SPMD view of `PTree(lt)`

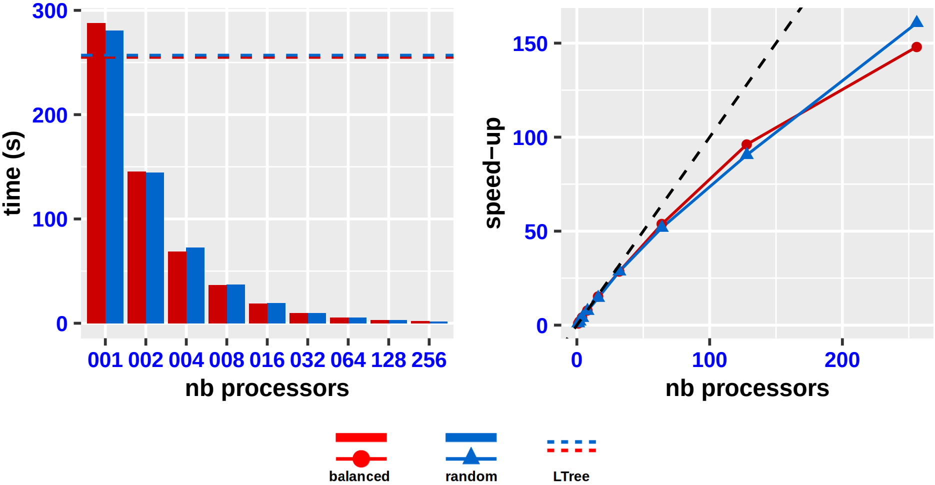
Example: Enumeration with prefix order

```
def prefix(input: PTree[A, B]) -> PTree[int, int]:
...
```

For a parallel implementation, need to use the following skeletons: `map`, `uacc`, and `dacc`.

Example: Enumeration with prefix order

Prefix on trees (balanced and random) of $2^{24} - 1$ elements.



HPC cluster (total of 24TB of memory), 16 Intel Xeon cores per node. Individual systems are interconnected via FDR Infiniband at a rate of 56Gbps. Ran 30 times with the following software: Ubuntu Linux 18.04, Python 3.6.7, mpi4py 3.0.0, OpenMPI

Write a better program

New challenge:

- Composition: How to write a program using the provided primitives?

Write a better program

New challenge:

- Composition: How to write a program using the provided primitives?

What is the best composition of skeletons?

Write a better program

New challenge:

- Composition: How to write a program using the provided primitives?

What is the best composition of skeletons?
$\Rightarrow$ Automatic programs rewriting

Write a better program

- Based on rewriting rules
- Aims at improving performances
- Implicit mechanism: keep the high-abstraction of PySke
- On lists only (for the moment)
- Innermost strategy (for the moment)

Implicit mechanism

In the first version of the API:

- Incremental execution (direct execution of calls)

In the new version:

- A computation tree is built and then ran as follows
  1. an optimization of the computation tree (application of rules with a innermost strategy; iteratively until no rules can be applied anymore)
  2. an execution of the composition corresponding to the new tree

A composition

```
data.meth1(args1).meth2(args2)
```

becomes

```
wrap(data).meth1(args1).meth2(args2).run()
```

Rewriting rules

Available rules:

- Optimization of composition of `mapss`
- Optimization of composition of `map` and `reduce`
  - Using `map_reduce` (internal skeleton that is more efficient)
  - Based on algebra (e.g., generalized De Morgan rules)
- Optimization using `curry`-ied and `uncurry`-ied functions

Rewriting rules

Available rules:

- Optimization of composition of mapss
- Optimization of composition of map and reduce
  - Using map_reduce (internal skeleton that is more efficient)
  - Based on algebra (e.g., generalized De Morgan rules)
- Optimization using curry-ied and uncurry-ied functions

Syntax (example):

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

Example: Dot product

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

- Direct implementation:

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

- Wrapped structures:

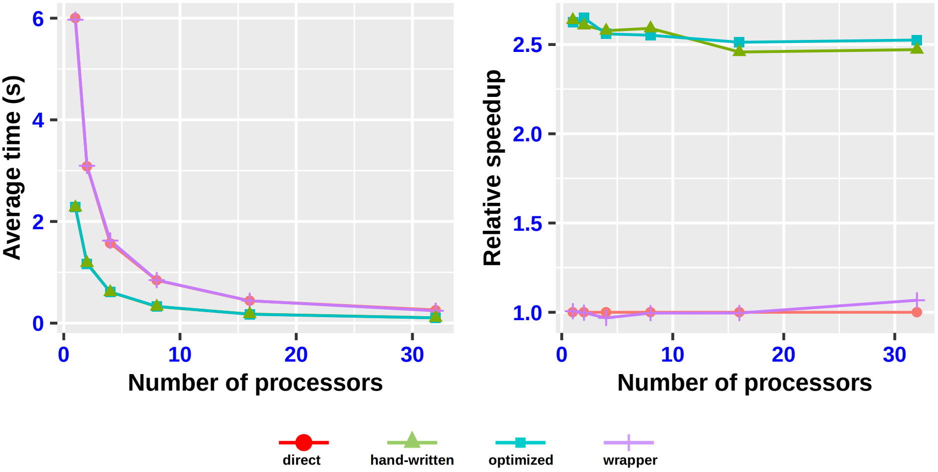
```
def dot_product_wrapped(pl1: PL, pl2: PL):
  pl1, pl2 = PList.wrap(pl1), PList.wrap(pl2)
  dot = pl2.zip(pl1).map(uncurry(mul)).reduce(add,
    0).run()
  return dot
```

- Hand-written optimal:

```
def dot_product_handwritten(pl1: PL, pl2: PL):
  return pl2.map2(mul, pl1).reduce(add, 0)
```

Example: Dot product

Dot product between lists of 5.10^7 elements (integers)



Shared memory machine (256 Gb), two Intel Xeon E5-2683 v4 (16 cores at 2.10 GHz).

Ran 30 times with the following software: CentOS 7, Python 3.6.3, mpi4py 3.0.2, OpenMPI 2.6.4.

Conclusion and Future Works

PySke : an API of Skeletons in Python

- A lot of skeletons on lists
- Tackle the lack of skeletons on Tree
- High-abstraction making parallelism accessible to every kind of users
- Automatic optimization mechanism

Conclusion and Future Works

PySke : an API of Skeletons in Python

- A lot of skeletons on lists
- Tackle the lack of skeletons on Tree
- High-abstraction making parallelism accessible to every kind of users
- Automatic optimization mechanism

And now?

- Other data-structures (e.g., graphs)
- More applications (e.g., clustering; graph and model transformation)
- Cost model of skeletons
- Optimization mechanism for all data structures

Publications

- J. Philippe. Systematic development of efficient programs on parallel data structures. (Master's thesis). At *School of Informatics Computing and Cyber Systems (SICCS)*. Northern Arizona University, May 2019.
- J. Philippe and F. Loulergue. PySke: Algorithmic skeletons for Python. In *International Conference on High Performance Computing and Simulation (HPCS)*. Dublin, Ireland: IEEE, Jul 2019.
- J. Philippe and F. Loulergue. Towards automatically optimizing PySke programs (poster). In *International Conference on High Performance Computing and Simulation (HPCS)*. Dublin, Ireland: IEEE, Jul 2019.
- F. Loulergue and J. Philippe. Automatic Optimization of Python Skeletal Parallel Programs. In *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*. Melbourne, Australia: Springer, Dec 2019.
- F. Loulergue and J. Philippe. New List Skeletons for the Python Skeleton Library. In *Parallel and Distributed Computing: Applications and Technologies (PDCAT)*. Gold Coast, Australia: Springer, Dec 2019.