Towards Automatically Optimizing PySke Programs

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

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

RESEARCH POSTER EXTENDED ABSTRACT

I. Parallel Skeletons in Python

Explicit parallel programming for shared and distributed memory architectures is an efficient way to deal with data intensive computations. However approaches such as explicit threads or MPI remain difficult solutions for most programmers. Indeed they have to face different constraints such as explicit inter-processors communications or data distribution.

PySke [1] is a Python library (on top of *mpi4py*) that aims at easing parallel programming for casual users. The implementation of parallel computation patterns, called skeletons [2], are provided by the library to keep abstract most of the parallel aspects of a program. Their implementations in Python are provided as high-order functions (i.e., functions that take other functions as input). This makes the skeletons/computation patterns very general.

PySke is currently a data-parallel approach: The skeletons operate on distributed data structures. However users do not have to explicitly manage the distribution of such data structures. Moreover, PySke provides equivalent computation patterns for sequential data structures and parallel data structures. We call these patterns *primitives* when they are implemented in sequential and *skeletons* when they are implemented in parallel.

PySke contains a class `SList` that extends the `list` class of Python with additional primitives, in a functional style (i.e. the primitives consider the data structure as immutable). It also contains a class `PList` that offers skeletons equivalent to the `SList` primitives. Trees are also part of PySke: rose trees, binary trees, linearized trees and distributed linearized trees. The same primitives and skeletons are provided for all these data structures.

Among the primitives and skeletons, `map` and `reduce`, inspired from functional programming, are part of PySke. The example shown in Figure 1 is a piece of PySke code that computes the Euclidean norm of a vector represented by `data`. This listing can be run either in sequential if `data` is an instance of `SList`, or in parallel if `data` is an instance of `PList`. This example also shows that PySke offers a *global view* of programs: the structure of the program is similar (and in this case identical) to a sequential program, but it operates on a parallel data structure. This is very different, and much more simple to understand for the casual programmers, from the SPMD paradigm where a program should be understood as the parallel composition of sequential communicating programs.

```
squared = data.map(operator.pow(x, 2))
summed = squared.reduce(operator.add)
norm = math.sqrt(summed)
```

Fig. 1. Example of PySke code: Euclidean norm

II. Program Transformations

A program can be optimized thanks to transformations: The goal is to reduce the computation costs. Calculi, such as the Bird-Meertens Formalism [3], [4], propose theorems and equalities that can considerably reduce the computation time of a program. These transformations range from avoiding multiple traversals of intermediate data structures to changing the order of algorithmic complexity of a program. The latter is of course more complex to achieve and is most often not performed automatically. The former can be automated yet can offer substantial performance gains.

For example, the composition of two map can be transformed into a single use of map, thus removing the allocation and transversal of an intermediate list. Considering two functions f and g, $(map\ f) \circ (map\ g)$ can be transformed into $map\ (f \circ g)$.

Other examples are more context-dependent and can take advantage of algebraic properties. For instance, Boolean values can be represented within a list, and handled with classical primitives such as map or $reduce$. Since the following equality holds: $\bigwedge_{i=1}^{n}(\neg\, x_i) = \neg(\bigvee_{i=1}^{n} x_i)$, the program $(reduce\ and) \circ (map\ not)$ can be optimized as $not \circ (reduce\ or)$.

These examples are equivalences between compositions of primitives, but they are also true if skeletons are used instead of sequential functions. There exist other forms of theorems which, from specific classes of functions, transform naive specifications into programs using primitives (e.g., the diffusion theorem [5], homomorphism theorems [6]).

In most of the existing skeletons libraries, these transformations are not automated. Some transformations exist for C++ libraries, than are based on meta-programming techniques. It is however quite complex to implement program transformations in this way, and only a few transformations are implemented in these libraries [7], [8]. In the context of functional pro-

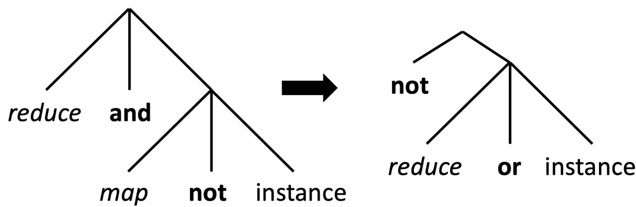

Fig. 2. Example of skeleton expression optimization

gramming, and the Coq proof assistant, such transformations are also automated [9].

PySke aims at easing parallel programming. The current version of the system only supports the composition of skeletons. Therefore we need to provide an automated way to optimize these compositions. Unlike C++ or Scala [10], but like [11], we propose to perform the transformation at runtime. In order to do so, the call to the skeletons will no longer directly call a given computation pattern, but will rather build a skeleton expression. This skeleton expression is then executed by explicitly calling a `run` method. This method will first optimize the skeleton expression using rewrite rules such as the equivalences mentioned above. For example, the skeleton expression corresponding to $(reduce\ and) \circ (map\ not)$ can be transformed as presented in Figure 2.

In the current version of the work, we do have the transformation rules we want to apply, but the transformations are applied manually. In the next version of PySke, we will provide automatic optimization at runtime.

III. Performances

To test how efficient is program transformation with PySke, we measured the computation times of three equivalent programs on distributed lists `X` of $5*10^7$ Booleans. Each program has been executed 30 times on 30 different datasets.

```
X.map(operator.not_).reduce(operator.and_)
X.map_reduce(operator.not_, operator.and_)
not (X.reduce(operator.or_))
```

In the second line, `map_reduce` is a skeleton we added. In a single pass it applies it first argument to the elements of the object list and computes the reduction. This skeleton has the same semantics than a composition of `map` and `reduce` but performs one less list traversal, and does not require the allocation of the intermediate list produced by `map`.

The three programs were executed on a shared memory machine (256 Gb), with two Intel Xeon E5-2683 v4 processors each having 16 cores at 2.10 GHz. The used software is the following: Ubuntu Linux 18.04, Python 3.6.7, mpi4py version 3.0.0, OpenMPI version 2.1.1. Figure 3 presents the average computation time, and the relative speed-up for each version. It demonstrates the interest of optimizing programs using transformation rules. In this case, we used specific rules, based on De Morgan's laws to obtain the best program. However, even by replacing the use of the two skeletons `map` and `reduce` by only `map_reduce`, we obtain better

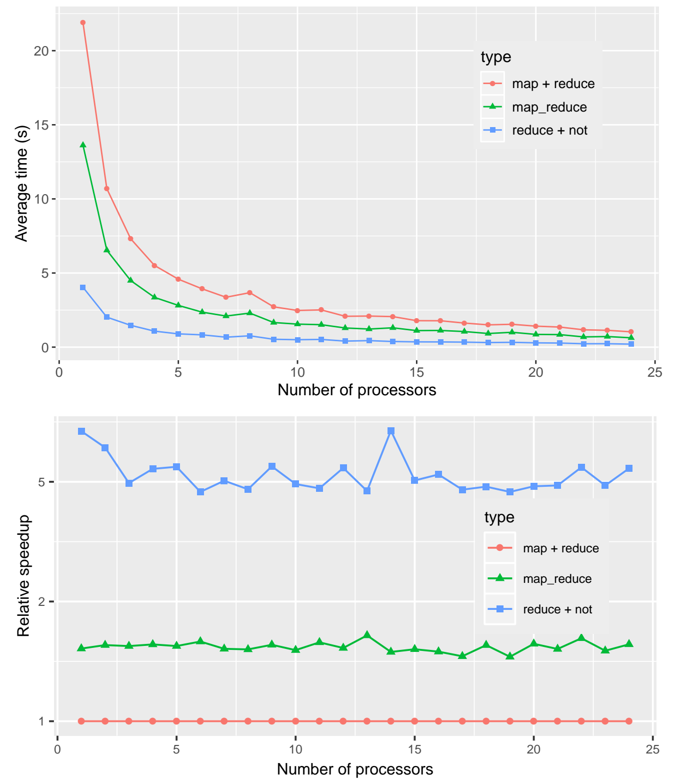

Fig. 3. Example of performance increases using program transformations

performances. This replacement can be done in all programs containing a composition of `map` and `reduce`.

References

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

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

[3] R. Backhouse, *Program Construction*. Wiley, 2003.

[4] R. Bird and O. de Moor, *Algebra of Programming*. Prentice Hall, 1996.

[5] 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.

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

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

[8] 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.

[9] F. Loulergue, W. Bousdira, and J. Tesson, “Calculating Parallel Programs in Coq using List Homomorphisms,” *Int J Parallel Prog*, vol. 45, pp. 300–319, 2017.

[10] 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, 2014.

[11] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, “Accelerating Haskell Array Codes with Multicore GPUs,” in *Workshop on Declarative Aspects of Multicore Programming (DAMP 2011)*. ACM, 2011.