Parallel Programming with Coq: Map and Reduce Skeletons on Trees

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

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

ABSTRACT

SyDPaCC is a set of libraries for the Coq interactive theorem prover. It allows to develop correct functional parallel programs on distributed lists based on the transformation of naive sequential programs that are considered as specifications. To offer the parallelization of functions on other data structures, the first step is to implement a parallel version of the considered data structure and to provide parallel implementations of primitive functions manipulating it. This paper presents such a first step: a binary tree extension which includes new map and reduce pure functional algorithmic skeletons for binary trees. Such algorithmic skeletons are templates of parallel algorithms, realized in a functional context as higher-order functions implemented in parallel. The use of these new primitives is illustrated on example applications.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; • **Software and its engineering → Functional languages**; *Formal methods.*

KEYWORDS

Functional programming; parallel programming; Coq.

ACM Reference Format:
Jolan Philippe and Frédéric Loulergue. 2019. Parallel Programming with Coq: Map and Reduce Skeletons on Trees. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3297280.3299742

1 INTRODUCTION

As most computing devices are now parallel architectures, parallel programming should be the norm. It remains difficult. A trade-off between programming productivity and execution efficiency is necessary, and this trade-off is dependent on the application domain. However, for mainstream applications, programming productivity is more important than pure execution efficiency.

To increase parallel programming productivity, one solution is to provide ready to use parallel patterns. It may restrict the set of parallel algorithms that can be expressed, and performances may be slightly lower than approaches such as threads or MPI. This approach is related to algorithmic skeletons [3] that are patterns of classical parallel algorithms. Skeletons, as Google's MapReduce, were inspired by functional programming. Most of the time, algorithmic skeletons are related to sequential patterns. For example, the map skeleton applies a function to all the elements of a collection of elements: this can be performed in sequential using an iterator on the collection, or in parallel if the collection is distributed.

From the user point-of-view, one challenge in skeletal parallelism is to choose the skeletons to use and to compose them. Being related to functional programming, skeletal parallelism is related to the theory of lists and more generally to the Bird-Merteens Formalism (BMF) [1]. BMF provides a methodology to obtain step-by-step an efficient functional program from an initial specification, either as an inefficient function or as a relation. It is also possible to use BMF to obtain parallel programs implemented using algorithmic skeletons [7].

SyDPaCC [9] written with the Coq proof assistant (https://coq.inria.fr), provides pure functional algorithmic skeletons based on lists and distributed lists. However, trees are useful data types in particular because they allow the representation of hierarchical structures. For example, they are used to represent structured documents (e.g., XML). Nevertheless, it is difficult to write efficient, and load-balanced parallel programs on trees: the tree structure is irregular, and not necessarily balanced.

To tackle these problems, Matsuzaki et al. have designed parallel skeletons for manipulating general trees. Their approach is based on a BMF formalization of trees, and provides primitives to handle both binary and rose trees [12]. Contrary to binary trees, rose trees are trees with nodes which have not necessary two children. Their skeletons are based on a simple scheme: arbitrary shaped trees are transformed into binary trees, binary tree skeletons are applied, and if it is necessary the result is re-transformed into a rose tree. From this central idea, they developed several approaches to compute in parallel on binary trees, e.g. [11].

Our goal is to extend SyDPaCC with distributed trees. A first step is to provide a parallel data structure representing binary trees and parallel algorithmic skeletons on this data structure. This is the contribution presented in this paper: a pure functional implementation of a distributed data structure of trees (Section 2) and a parallel implementation of skeletons manipulating it, with the Coq proof assistant (Section 3). Moreover, we performed performance tests on two applications obtained as OCaml code extracted from Coq, with calls to the parallel functional programming library BSML [10] (Section 4). We present related work in Section 5 and conclude in Section 6.

ACM ISBN 978-1-4503-5933-7/19/04.
https://doi.org/10.1145/3297280.3299742

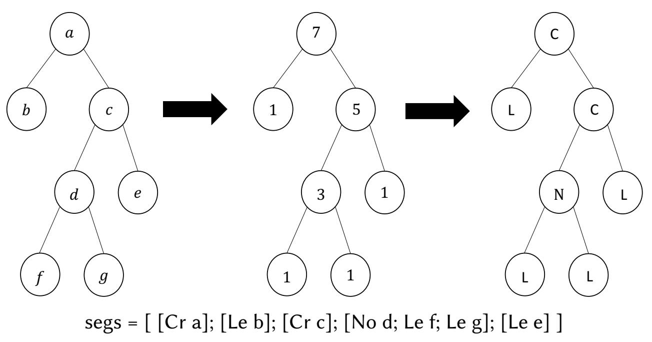
a
b
c
d
e
f
g
7
1
5
3
1
1
1
C
L
C
N
L
L
L
segs = [[Cr a]; [Le b]; [Cr c]; [No d; Le f; Le g]; [Le e]]

Figure 1: An example of a list representation of a tree (m=3)

2 A LIST REPRESENTATION OF TREES

We write $f\ x$ for the application of function f to element x, instead of $f(x)$. Parenthesis are still needed in some cases: for example, we write $f\ (g\ x)$ to apply function f to the result of the application of g to x, while $f\ g\ x$ means that function f has two arguments, g and x respectively. $\lambda x \Rightarrow e$ denotes an anonymous function that has an argument x and a body e. $\circ$ denotes function composition.

A list is a homogeneous sequence of values of the same type. A list has type *list* α when its elements have type α. A list can either be empty, written [], or $x :: xs$ where x is the head element and xs a tail list. $[x_1;\ x_2]$ is another notation for $x_1 :: x_2 :: []$. Informally, applying a function f to each element of a list is defined as:

$$map\ f\ [x_1;\ x_2;\ \ldots;\ x_n] = [f x_1;\ f x_2;\ \ldots;\ f x_n]$$

Considering an associative operation $\oplus$ and its neutral element $i_\oplus$, *reduce* can be informally defined as follows:

$$reduce\ (\oplus)\ [x_1;\ x_2;\ \ldots;\ x_n] = i_\oplus \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_n$$

In the Coq code, these two functions will be denoted respectively by List.map and List.reduce.

Contrary to lists, the structure of trees is not linear. We describe here the structure of binary trees. There are two ways of building a tree: *Leaf a* represents a leaf containing a value a, and *Node* $(l,\ b,\ r)$ represents a node built using a value b and two subtrees l and r. The type of the values contained in the leaves are not necessarily the same than the ones in the nodes: a tree of type *BTree* $\alpha\ \beta$ has leaf values of type α and node values of type β.

map on trees is defined as follows:

$$\begin{cases} map\ k_L\ k_N\ (Leaf\ a) & = \quad Leaf\ k_L\ a \\ map\ k_L\ k_N\ (Node\ (l,\ b,\ r)) & = \\ \qquad Node\ (map\ k_L\ k_N\ l,\ k_N\ b,\ map\ k_L\ k_N\ r) & \end{cases}$$

The reduction of a tree is defined as:

$$\begin{cases} reduce\ k\ (Leaf\ a) & = \quad a \\ reduce\ k\ (Node\ (l,\ b,\ r)) & = \quad k\ (reduce\ k\ l)\ b\ (reduce\ k\ r) \end{cases}$$

To develop skeletons to compute on trees on distributed-memory parallel computers, we must split the data structure into smaller elements which will be distributed. It is easy to do for a list of n elements: if we have p processors we divide the list into p sub-lists of about $\frac{n}{p}$ elements. It is more difficult for trees. One solution is first to obtain a sequential linear structure representing trees. The linearization used here is based on the *m-Bridge* algorithm, defined from graph theory [13], which consists in splitting the tree into several subtrees and transforming these subtrees into lists.

```
Inductive val A B := | Le: A→val A B | No: B→val A B | Cr: B→val A B.
Definition segment A B := list (val A B).

Definition map_local A B C D (kL: A→C)(kN:B→D)(seg: segment A B) :=
 let fmap (v:val A B) :=
   match v with|Le v⇒Le(kL v) |No v⇒No(kN v) |Cr v⇒Cr(kN v) end
 in List.map fmap seg.

Definition map A B C D (kL: A→C)(kN:B→D)(segs:list(segment A B)) :=
 List.map (map_local kL kN) segs

Definition reduce A B C (k: (A*B*A) → A)(segs : list (segment A B))
 {Hc : Closure A B C k phi psiN psiL psiR} : option A :=
 let local := map_filter_some (reduce_local k phi psiL psiR) segs in
   reduce_global psiN local.
```

Figure 2: Coq Code: Sequential Functions

Definition 2.1 (m-Critical Node)**.** Giving an integer m, a node t is called m-critical if for each t' child of t, the following inequality is respected: $\lceil size(t)/m \rceil > \lceil size(t')/m \rceil$, where $size(t)$ denotes the number of elements in the tree t.

Definition 2.2 (m-Bridge)**.** Giving an integer m, a m-bridge is a set of adjacent nodes divided by m-critical nodes. In other words, a m-bridge is a set of adjacent nodes where the m-critical nodes are only root or bottom of the resulting subtrees.

The critical nodes are the cut points of the tree. According to definition 2.2, they are either the root or the bottom of the resulting subtrees.

In a linearized tree, data is stored in lists of values, called segments. Figure 2 presents the Coq definition of a segment. It is a list of values, each being either a leaf value, a node value, or a critical node value. An example is given in Figure 1 for $m = 3$.

map and *reduce* can be defined on linearized trees.

The *map* function is quite easy to define since there is no dependency among the nodes during the computation. A function *map_local* is defined which will be applied to each segment. It aims to apply k_L (resp. k_N) on the elements marked as leaves (resp. as nodes or critical nodes). The function is defined in Figure 2.

Contrary to *map*, *reduce* needs for each node to combine the results obtained from reducing its children trees with the value held by the node as shown before.

For a linearized tree, that is a list of segments, it means we first need to reduce each segment, then reduce the intermediate list of partial reductions. The reduction of each segment is also not as direct as the reduction on binary trees because it is more difficult to identify the children of a node in a segment.

Computing a reduction on a linearized tree requires that its parameter function k verify a property named the closure property: the existence of four auxiliary functions ϕ ψ_n ψ_l and ψ_r such that:

$$\begin{cases} k\ l\ b\ r & = \quad \psi_n\ l\ (\phi\ b)\ r \\ \psi_n\ (\psi_n\ x\ l\ y)\ b\ r & = \quad \psi_n\ x\ (\psi_l\ l\ b\ r)\ y \\ \psi_n\ l\ b\ (\psi_n\ x\ r\ y) & = \quad \psi_n\ x\ (\psi_r\ l\ b\ r)\ y \end{cases} \tag{1}$$

Intuitively this property corresponds to a generalization of the associativity for a binary operation.

The reduction on a linearized tree proceeds in two steps. First, *reduce_local* applies the functions ϕ and either ψ_l or ψ_r to the m-critical nodes and their ancestors, and apply k to the other internal nodes. Each segment will then become a single value, corresponding to a local reduction. Nonetheless, the local reduction, as described, only works if it is applied to a segment obtained as the correct linearization of a tree. For a list of values that is not well-formed, this function returns None. For a well-formed segment it returns a value Some v where v is the result of the local reduction. To reduce the list of segments, we could apply List.map to reduce_local and the linearized tree. However, we need to filter out the None results and transform the values of the form Some v into just v. This is done by a function named map_filter_some. For example, if a function f returns None of all odd number and Some n for any even number n, map_filter_some f [1;2;3;4] returns [2;4].

In the second step, all these intermediate results are merged using another function, *reduce_global*, into a single value thanks to ψ_n. The sequential version in Coq is shown in Figure 2. Note that in this code, the argument Hc is a pre-condition: it means the function k should satisfy the closure property (1).

3 PARALLELIZATION: TREE SKELETONS

Bulk Synchronous Parallel ML (BSML) is a functional parallel programming *language* currently implemented as a *library* for the OCaml language [10] that follows the BSP model [15]. A BSP computer is a set of processor-memory pairs, connected by a network for point-to-point communications, and a global synchronization unit. The execution of a BSP program proceeds as a sequence of super-steps divided into three phases. In the computation phase, each processor computes using only the data it holds in its private memory. The communication phase processes data exchange. After the last phase, the synchronization, the exchanged messages are guaranteed to have reached their destinations. BSML offers a parallel data structure: for a type A, par A is a parallel vector of elements of type A. Each processor holds only one value of type A. In the BSP model the number of processes remains the same during all execution: thus the size of parallel vectors is fixed. The nesting of such vectors is not allowed.

In our implementation of skeletons, the considered data structure will be a parallel vector of lists of segments. It is possible to think sequentially about this data structure as the concatenation of the lists of segments.

As described in the previous section, on a linearized version of a tree, the *map* function on trees is a *map* function on lists. Moreover, the SyDPaCC framework already features a *map* skeleton on distributed lists, named ParList.map. A parallel *map* on distributed trees is shown in Figure 3.

The parallel version of reduction proceeds, as the sequential version, in two steps. For the first step, instead of having one list of segments, we have one list of segments per processor. So we need to apply the first step of the sequential version at each processor. That is done using the parfun: ∀ A B, (A→B)→par A→par B function of BSML. Intuitively parfun transforms a sequential function into a function that operates on parallel vectors, applying the sequential

```
Definition map_par {A B C D} (kL:A→C) (kN:B→D) :
  par (list (segment A B)) → par (list (segment C D)) :=
 ParList.map (map_local kL kN).

Definition reduce_par A B C (k: (A*B*A)→A) (v:par(list(segment A B)))
 {Hc: Closure A B C k phi psiN psiL psiR} : option A :=
 let l:=parfun(map_filter_some(reduce_local k phi psiL psiR)) v in
 reduce_global psiN (ParList.join local).
```

Figure 3: Coq Code: Tree Skeletons

function in parallel at each processor. In Figure 3, the result of this first step is named l that has type par (list(sum A C)). sum A C means that each element of the list has type A or type C.

The second step cannot be made in parallel. As the result of the previous step is a parallel vector, we first need to transform this parallel vector into a list. This is done by the SyDPaCC function ParList.join that transforms a parallel vector of lists into a list (present on all the processors). We perform the global reduction on the resulting list.

4 APPLICATIONS AND EXPERIMENTS

We present here two examples of the use of *map* and *reduce* on distributed linearized trees and experiments with them on a parallel machine.

The height of a binary tree can be written with a recursive function as a single-bottom up computation:

$$\begin{cases} height\ (Leaf\ a) & = & 1 \\ height\ (Node\ (b,\ l,\ r)) & = & 1 + (l \uparrow r) \end{cases}$$

where $l \uparrow r$ computes the maximum of l and r. This function can be easily defined thanks to the *map* and *reduce* functions:

$$height = (reduce\,(\lambda(x,l,r) \Rightarrow x{+}(l \uparrow r))) \circ (map\,(\lambda\,x{\Rightarrow}1)(\lambda\,x{\Rightarrow}1))$$

For a parallel version, we need to use map_par and reduce_par. For reduce_par we need to prove that the function argument of reduce satisfies the closure property. It is the case with:

$$\begin{cases} \phi\ b & = & (-\infty, b) \\ \psi_N\ l\ (b_1,\ b_2)\ r & = & b_1 \uparrow (b_2 + l) \uparrow (b_2 + r) \\ \psi_L\ (l_1,\ l_2)\ (b_1,\ b_2)\ r & = & (b_1 \uparrow (b_1 + l_1) \uparrow (b_2 + r), b_2 + l_2) \\ \psi_R\ l\ (b_1,\ b_2)\ (r_1,\ r_2) & = & (b_1 \uparrow (b_2 + l) \uparrow (b_2 + r_1), b_2 + r_2) \end{cases}$$

The *properties count* application aims at getting the number of elements which respect a given property. Considering two predicates $p_L : \alpha \to bool$ and $p_N : \beta \to bool$ the number of leaves and nodes respecting the properties in a binary tree of type *BTree α β* can be implemented as a single-bottom up computation:

$$\begin{cases} count\,p_L\,p_N\ (Leaf\ x) \quad = \ if\ (p_L\ x)\ then\ 1\ else\ 0 \\ count\,p_L\,p_N\ (Node\ (l,\ x,\ r)) = \\ (if\ p_N\ x\ then\ 1\ else\ 0) +\ (count\,p_L\,p_N\ l) +\ (count\,p_L\,p_N\ r) \end{cases}$$

The function can be written using the primitives *reduce* and *map*:

$$count\,p_L\,p_N = (reduce\ (\lambda(x,l,r) \Rightarrow x + l + r))\ \circ\ (map\ f_L\ f_N)$$
$$\textbf{with}\ f\ p\ x = if\ p\ x\ then\ 1\ else\ 0\ \textbf{and}\ f_L = f\ p_L\ \textbf{and}\ f_N = f\ p_N$$

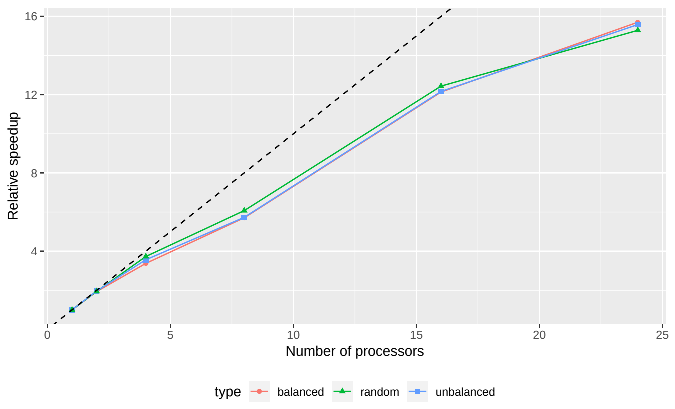

Figure 4: Relative Speed-up

and the closure property holds for:

$$\begin{cases} \phi\ b = b & \psi_N\ l\ b\ r = b + l + r \\ \psi_L\ l\ b\ r = b + l + r & \psi_R\ l\ b\ r = b + l + r. \end{cases}$$

Experiments were conducted on a shared memory machine with two Intel Xeon E5-2683 v4 processors with 16 cores at 2.10 GHz, 256Gb of memory. The following resources were used: Ubuntu Linux 18.04, Coq version 8.8.1, BSML version 0.5.5, OpenMPI version 1.10.7, and OCaml version 4.02.3.

The tests have been conducted using the *count* function, on a tree of 3×3 matrices of size $2^{23} - 1$. The application counts the number of orthogonal matrices. The value of m used to split the tree is calculated by $m = 2\sqrt{N}$ with N the size of the tree. 30 measures have been taken on three kinds of tree: balanced, completely unbalanced and with a random shape. Figure 4 shows the average relative speedup for each type of tree depending on the number of processors p. These experiments show that the obtained performances does not depend heavily on the kind of tree.

However, the value of m has importance. A small value creates a list of small segments which is easier to distribute in a balanced way but leads to more costly communications during reduction. A large value makes fewer segments and leads to cheaper communications, but the distribution may be more unbalanced. The current value of m does not depend on the number of processors. We plan to take into account the BSP parameters of the parallel machine as well as parameters specific to the application (basically the size of each element of the tree, assumed the same for all elements, and the time required to test the elements, assumed constant) to analytically have a bound for the best values of m.

5 RELATED WORK

On the implementation side, algorithmic skeletons libraries mostly consider linear data structures such as lists and arrays [2, 5, 8]. One exception is SkeTo (http://sketo.ipl-lab.org): its earlier versions contained tree algorithmic skeletons, but the latest version does not, although recent work considers a new implementation [14].

To our knowledge, the proposed implementation of binary trees is the only pure functional explicit parallel implementation. Being implemented using a proof assistant is another distinctive feature.

However proof assistants have been used to reason about parallel programs. For example, Grégoire and Chlipala provide a small parallel language and its semantics and proves correct optimizations of stencil-based computations [6]. A subset of Data Parallel C has been formalized using the Isabelle/HOL proof assistant [4]. None of these works consider trees.

6 CONCLUSION AND FUTURE WORK

We have extended the SyDPaCC framework with a new set of algorithmic skeletons for tree manipulation on a distributed data structure of linearized trees. This allows to write parallel programs on trees within Coq and to reason about them. The full source code is available at https://sydpacc.github.io.

As future work, we plan to extend SyDPaCC so that its verified automatic parallelization feature can target this new data structure and skeletons. To do so we need to provide *verified* type correspondences first between binary trees and linearized trees, and then between linearized trees and parallel linearized trees. Based on these correspondences, we will prove the correspondence of sequential map and reduce on trees with sequential map and reduce on linearized trees, and then with parallel map and reduce on parallel linearized trees.

REFERENCES

[1] Richard Bird and Oege de Moor. 1996. *Algebra of Programming*. Prentice Hall.
[2] Philipp Ciechanowicz and Herbert Kuchen. 2010. Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*. 108–113. https://doi.org/10.1109/HPCC.2010.23
[3] Murray Cole. 1989. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press. Available at http://homepages.inf.ed.ac.uk/mic/Pubs.
[4] M. Daum. 2007. Reasoning on Data-Parallel Programs in Isabelle/Hol. In *C/C++ Verification Workshop*. https://doi.org/~rhuuck/CV07/program.html
[5] Roberto Di Cosmo and Marco Danelutto. 2012. A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml. In *International Conference on Computational Science (ICCS)*, Vol. 9. Elsevier, 1837–1846. https://doi.org/10.1016/j.procs.2012.04.202
[6] Thomas Grégoire and Adam Chlipala. 2018. Mostly Automated Formal Verification of Loop Dependencies with Applications to Distributed Stencil Algorithms. *Journal of Automated Reasoning* (2018). https://doi.org/10.1007/s10817-018-9451-y
[7] Zhenjiang Hu, Hidewaki Iwasaki, and Masato Takeichi. 1997. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Trans Program Lang Syst* 19, 3 (1997), 444–461. https://doi.org/10.1145/256167.256201
[8] Joeffrey Légaux, Frédéric Loulergue, and Sylvain Jubertie. 2013. Managing Arbitrary Distributions of Arrays in Orléans Skeleton Library. In *International Conference on High Performance Computing and Simulation (HPCS)*. IEEE, Helsinki, Finland, 437–444. https://doi.org/10.1109/HPCSim.2013.6641451
[9] Frédéric Loulergue, Wadoud Bousdira, and Julien Tesson. 2017. Calculating Parallel Programs in Coq using List Homomorphisms. *Int J Parallel Prog* 45 (2017), 300–319. Issue 2. https://doi.org/10.1007/s10766-016-0415-8
[10] Frédéric Loulergue, Frédéric Gava, and David Billiet. 2005. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In *International Conference on Computational Science (ICCS) (LNCS)*, Vol. 3515. Springer, 1046–1054. https://doi.org/10.1007/11428848_132
[11] Kiminori Matsuzaki. 2017. Efficient Implementation of Tree Skeletons on Distributed-Memory Parallel Computers. *Scalable Computing: Practice and Experience* 18, 1 (2017), 17–34. https://doi.org/10.12694/scpe.v18i1.1231
[12] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2006. Parallel Skeletons for Manipulating General Trees. *Parallel Comput.* 32, 7 (Sept. 2006), 590–603. https://doi.org/10.1016/j.parco.2006.06.002
[13] J. H. Reif (Ed.). 1993. *Synthesis of Parallel Algorithms*. Morgan Kaufmann.
[14] Shigeyuki Sato and Kiminori Matsuzaki. 2016. A Generic Implementation of Tree Skeletons. *Int J Parallel Prog* 44, 3 (2016), 686–707. https://doi.org/10.1007/s10766-015-0355-6
[15] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103. https://doi.org/10.1145/79173.79181