SYSTEMATIC DEVELOPMENT OF EFFICIENT PROGRAMS ON PARALLEL DATA STRUCTURES

By Jolan Philippe

A Thesis
Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in Computer Science

Northern Arizona University
April 2019

Approved:

Frédéric Loulergue, Ph.D., Chair
Hélène Coullon, Ph.D.
Frédéric Dabrowski, Ph.D.
Michael Gowanlock, Ph.D.

Abstract

Title:

Systematic development of efficient programs on parallel data structures

Jolan Philippe

Abstract: The continually increasing size of data implies the need for better computing techniques. Modern computers are parallel architectures containing several processors. However, writing parallel programs is not an easy task. Some of the algorithms designed for sequential data structure can be parallelized using the skeletal parallelism approach. Algorithmic skeletons are patterns of parallel algorithms, often realized as high-order functions implemented in parallel, manipulating distributed data structures. This thesis presents first the Bird-Meertens Formalism, used as a foundation for parallel programming. Secondly, we present the design of PySke, a library of skeletons implementation in Python. To ensure correctness, some of these skeletons are formalized and proved correct in the proof assistant Coq. Correct parallel examples are written with Coq and extracted to be run with BSML, a functional parallel programming implementation of the bulk synchronous parallel model.

Keywords: Functional programming; parallel programming; bulk synchronous parallelism; formal verification; interactive theorem proving; algorithmic skeletons; Python; Coq.

Acknowledgments

> You can be anything you want to be, just turn yourself into anything you think that you could ever be
>
> *Innuendo*
> ***Freddie Mercury***

I first want to thank Dr. Hélène Coullon and Dr. Frédéric Dabrowski for having been members of my committee despite the time difference. It has been an honor, and I hope to meet with you again in professional, or personal life. I would also like to thank Dr. Michael Gowanlock, who also have been a part of my jury, for his support and his expertise on parallel programming. I had the great opportunity of attending his classes and the chance of working with him. His always constructive suggestions have improved mainly my research skills. Thank you again for your feedback on this present thesis. To the last member of my jury Pr. Frédéric Loulergue, I would like to say a big thank you for offering me the opportunity of studying abroad, and for advising me through my research studies. We cannot expect better from someone to help you during the long journey of a researcher life. Thank you very much for continually pushing me to be better. Anything would be possible without you, and I hope to have opportunities to work with you again.

I must express my sincere gratitude to my dear Benoit Gallet, who has been the first who believed in me. He kept encouraging me for many years now, and I probably would not be in the same position without him. To my best friend, Anthony Neau, for all these incredible moments spent together, I address the most sincere thank you. Even if we took different directions, I am glad we are both following our dreams. I guess we deserve it. I wish all the best in your career. I miss you. I address a special thanks to my friend and colleague Salwa Souaf, for providing me wisdom, advice, and step back as often as it is necessary. An

additional thank you to Pierre Estienne, for his continued interest in my work, and his friendship in all circumstances.

Other special thanks to Galina Huard, Damien Chedeville, Julie Colson, Younes Bouayadine, Marie Agnes Villoue, and Bertrand Orain. None of this journey would be possible without your support. I am grateful to my parents, who have provided me the support to embark on this incredible adventure. I have nothing to had, you already know. I hope you are proud of me.

Finally, a big thank you to my girlfriend Camille Labonne, for her love and support. I dedicate my thesis to this incredible and lovely person.

Contents

List of Figures

Acronyms

ALU	Arithmetic Logic Unit
BMF	Bird-Meertens Formalism
BSML	Bulk Synchronous parallel ML
CiC	Calculus of inductive Constructions
CoC	Calculus Of Constructions
CPU	Central Processing Unit
CRCW	Concurrent Read Concurrent Write
CREW	Concurrent Read Exclusive Write
DSL	Domain Specific Language
EREW	Exclusive Read Exclusive Write
GPU	Graphics Processing Unit
GTA	Generate Test Aggregate
HOL	High Order Logic
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MPI	Message Passing Interface
MPS	Maximum Prefix Sum
PRAM	Parallel Random Access Machine
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
SISD	Single Instruction Single Data

Introduction 1

Contents

The raise of informatics led to important data analysis challenges especially about treating large-scale data structures as fast as possible, without negatively influencing the quality of the results. One solution is to use parallel computations to process several operations at the same time. However, this approach remains difficult: it is error-prone, and not easy to write. These programs aims at exploiting parallel architectures, but for that, they need to be explicitly parallel.

In this thesis, we try to answer these challenges. Our purpose is to give a solution to remove errors from parallel programs. More generally, we also try to answer the question: How to ease parallel programming for all developers?

In this chapter, we first introduce the advantages but also the challenges of parallel programming (Section 1.1). One difficulty is to maintain a parallel program correct. There exist different techniques presented in Section 1.2. Finally, we describe our contributions to tackle some of the difficulties of a parallel approach (Section 1.3).

Parallel programming

Nowadays, informatics is everywhere: on your computer, on your smartphone, even where you do not expect like your TV and your car. Many processes are executed in all your devices, allowing automatic systems and faster decisions. From devices, pieces of information are collected to be exploited (e.g., statistical analysis). However, since digital devices are increasingly used, the quantity of data upsurges. Capturing, curating, and managing huge amount of data within a decent elapsed time is a challenging task.

The science of large-scale study is called Big Data. It represents a relative amount of data around key concepts: *volume*; how to treat huge amount of data; *variety*; the data are from different sources not necessarily structured or organized; and *velocity*; how to reach a decent speed of computation. The word *relative* is used because the quantification behind *Big* varies depending on users, and the tools they have. For example, reading a terabyte of data with an old machine, with slow resources, is more challenging than doing the same operation with a very modern computer. Nonetheless, performance expectations must be realistic. For sure, we cannot expect that the old machine performs this operation in a few seconds but increasing the speed of calculation remains a Big Data problem. The term *Big* can be then understood as *to large*, with a relationship between resources, the size of data, and possible improvements.

Once the data are organized, they are stored into different data structures that are numerous (e.g., arrays, lists, graphs or trees). The chosen one depends on the properties we want to maintain (e.g., hierarchy in a tree). Although having organized data helps for their comprehension, their treatment remains a hard task. Naively, very inefficient programs can be written but a challenge behind Big Data is to compute as fast as possible. We need to find better approaches with related implementations and well-suited architectures to meet this challenge.

The evolutions of hardware and software have been related since the beginning of computer science. New hardware components can improve the performance of software. Continuous improvement of hardware is therefore necessary. One basic idea to increase computation speed is to use several processing units. It leads to parallel computing, a kind of computation which processes several calculations at the same time. It aims at getting faster results. The comparison between the performances of a sequential program and its parallel counterpart can be evaluated with mathematical formula [3, 60]. In theory, multiplying the number of processors by a value for a task should divide the computation time

by the same value. It is almost never true, especially when inter processing units communications are needed.

Parallel computers have existed from the early ages of computing but the first ones were not designed for a personal use. It is thanks to the apparition of multicore microprocessors that the personal computers are now able to do parallelism. This technology is henceforth pervasive, and all the processors are now parallel.

In this thesis, we aim at targeting large-scale applications that require large-scale machines both in terms of processing units, but also in terms of memory.

Architectures and models

In the modern computer science world, system architectures are described using Flynn's taxonomy [48]. It represents computer architectures with four approaches:

- **Single Instruction Single Data (SISD)** corresponds to a uniprocessor model. The data are treated one by one, in sequential order. This architecture is also known as the Von Neumann architecture [124].
- In a **Single Instruction Multiple Data (SIMD)** architecture, one operation is applied to several data. Most of the modern processors use SIMD by a vectorial approach. **Single Program Multiple Data (SPMD)** is a variant of this approach. It is very similar so that these two approaches are not mutually exclusive. SPMD is a much higher level of abstraction. The processors operate on different subsets of the data, but different operations may be applied at the same time.
- **Multiple Instruction Single Data (MISD)** applies successive treatments to data. This category is usually associated with pipelines and numerical filtering.
- **Multiple Instruction Multiple Data (MIMD)** is the most used parallel architecture. It is composed of calculation units with their data to treat. In other words, treatments are entirely independent. This architecture can be used with two types of memory:
  - Different programs at the same time can access the **shared memory**. Languages provide libraries to do shared memory parallelism, such as

the Pthreads library in C [110]. This architecture has, however, limitations. All the Central Processing Unit (CPU) cores access to the memory with a shared bus, which represents an obvious concurrency issue. Critical sections must be defined, and the memory accesses must be restricted.

– With a **distributed memory**, each processor has its private memory and is the only which can access to its data. Concrete communications must operate the exchange of information between processors (e.g., using Message Passing Interface (MPI)). More details about programming with a distributed memory are given in Chapter 4.

In this thesis we target distributed memory machines for large-scale applications, even though smaller scale shared memory systems are also of interest since they are easy to use.

Modern computers contain all of these architectures but at different levels of their overall architecture.

Parallel programing: An imperfect approach

Parallel approaches sound to be a good strategy for Big Data problems. It is now mainstream, and through language libraries, every developer can take advantage of the full resources of a machine. However, it remains a complex solution. First of all, it is not natural for programmers to think with a parallel approach. To scale a parallel program, several constraints must be considered: the distribution of the data, the dependence between the processes, the communication in a distributed memory system, the access to shared data (critical sections) in a shared memory system, and so on. On a sequential program, every instruction is surrounded by a prior and a consequent machine state. With the non-determinism of parallel programming, it is harder to predict the state of a machine at a specific time. The consequence of the difficulty to write parallel programs is a lack of parallel programmers and programs remain error-prone.

There exist compilers that can do automatic parallelization of sequential programs. However, because the compiler must be designed for a vast spectrum of cases, the resulting parallel programs can lose efficiency compared to handmade parallel programs. (e.g., nested loop structures with statically determined iteration counts).

How to write correct programs

Error detection is a part of integrated development environments (IDEs) or compilers. By analyzing the source code, these tools can detect problems such as non-initialized variables, type errors, or the use of non-existing functions. However, runtime errors cannot be identified as easily. All C programmers have met the `Segmentation Fault` error and looked for the cause of the failure during hours to finally fix it.

Before starting programming, program requirements must be defined on a document called the specification. A correct program is a program that respects its specification; that is the program exactly does what it is expected to do.

Correctness is then a representation of a quality level. A weak notion of correctness is the absence of runtime errors, and a strong one is that a program always returns expected output. A program is never perfect in the first draft. Indeed, small errors that are only detected after the first execution (e.g., index errors) are often made. Even experienced programmers must debug their algorithms. There exist several methods for bug detection with associated techniques and tools.

The first method, called testing, presented in section 1.2.1 is used at different levels of a program conception to ensure that the program is correct for particular cases. It checks if, for specific parameters, a program returns the expected result. There are many mathematically-based and logic-based methods to improve the quality of software, referred as formal methods. The detailed methods in this thesis aim to verify that properties hold for all possible executions. This verification can be divided into two categories: *a posteri verification* and *correctness by construction*. They are respectively presented in section 1.2.2 and section 1.2.3.

Tests and Quality

The functional specification of a program characterizes the output produced for all possible inputs. More precisely, the program behavior must respect a series of requirements, that can be formal (defined by mathematical terms) or informal (text in a natural language and possible graphical information in an informal or a semi-formal notation). The expected behavior may be even more specific with, for example, a description of the complexity and the efficiency of the program.

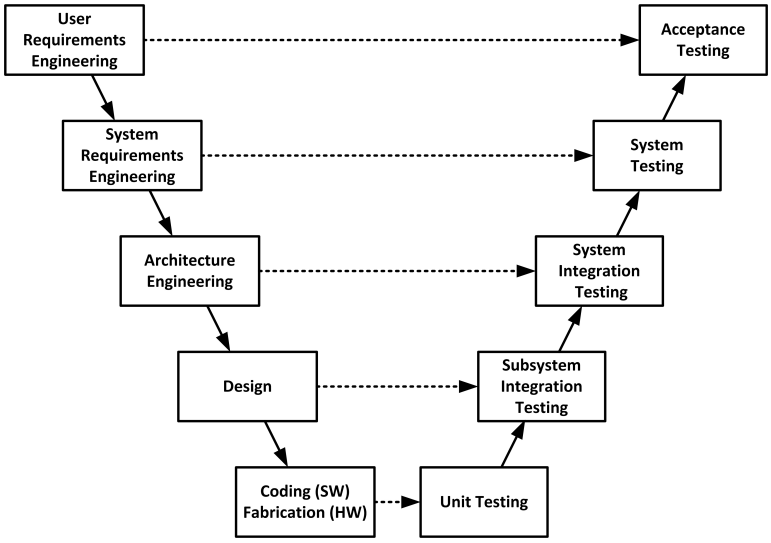


Figure 1.1 – *Traditional waterfall model for software conception: the V-model [46]*

V-model

In industry, the test technique has evolved through the years. The V-models (sometimes written V-modells) are very old-fashioned waterfall models [46]. For each step of conception, the work product is tested following different and specific techniques: analysis, demonstration, inspection, and finally testing. V-models are composed of two sides:

The left one is a descending side which describes the users needs with different levels of precision into small pieces. Concerning the right one, the corresponding tests are defined according to the tested pieces. The programming part is determined by the coding section and its associated unit tests. Each function, or procedure, is individually tested with different inputs. The results of the tests are a set of a couple of input-output. Regarding these results, we can accept, or reject, that the program respects its specification. Figure 1.1 from [46] illustrates this model.

The classical V-model has been derived into three other models, giving more details about each production step:

- **The single V-model** represents the work products rather than developed activities.
- **The double V-model** specifies the type of tests for each conception step.
- **The triple V-model** describes the importance of each acceptance step.

The main cons with tests are their specificity. On one hand, it is straightforward and natural to write and execute tests. On the other hand, it only verifies the quality of the program for specific cases. In other words, tests do not cover all the possibility of execution. According to Edsger W. Dijkstra [16],

> *Program testing can be used to show the presence of bugs,*
> *but never to show their absence.*

Formal methods will be preferred to verify the correctness of a program for all possible inputs and execution.

Modern software development techniques

In modern software developments, different methods are used to get quality controlled applications. They are classified as Agile methods and are based on a systems development life cycle. The work in Agile methods is based on iterations of small increments that minimize the amount of planning and design devices of an application. Contrary to waterfall models, where the build phase and the testing phase are separated, the development testing is completed in the same iteration as programming. That is, an increment is both a software component and related tests. More specifically, the test-driven development consists of writing the tests focused on requirements before writing the code. Another Agile approach relative to tests is continuous integration. With this practice, all the tests are run at each modification of the source code to check the absence of regression.

A posteri verification

In *a posteri verification*, the specification and the program are written independently. When they are both finished, a proof of correspondence is made to ensure the correctness of the program. The usual approach for doing this kind of verification is by using a deductive system, also called deductive inference. It consists of the use of axioms, or inference rules, defined in a semantic [111] to prove properties at a specific moment of the execution of a program. The most used formal system is the Hoare logic [66, 67], inspired by Floyd's works on flowcharts [47]. Hoare logic is based on Hoare triples describing a state of the computation. They are defined by $\{P\}C\{Q\}$ where P and Q are assertions and C a command to execute. P are the valid assertions before the execution of C and are called the preconditions while Q are the valid assertions after, and are called the postconditions. This logic can be naturally applied to most of the sequential

imperative programs. There exist tools to support this kind of verification for the languages, such as Frama-C for sequential C Programs [119].

By construction

In *correctness-by-construction*, the specification is written first and then transformed step-by-step into an efficient executable program. Each transformation is proved correct. In other words, the previous and the new models are shown equivalent. For example, the Bird-Meertens Formalism (BMF) (also called Squiggol) [6, 9, 52, 105] is a calculus that provides rules of equivalence between standard primitives on data structures to get a more efficient program. Program calculation, in particular of functional programs [8, 54], is a style of reasoning of *correctness-by-construction*. Proofs assistants [118] are well suited to conduct program calculation reasoning on functional languages [125]. The Coq [126] framework SYDPACC [92] has been designed for *correctness-by-construction* with a skeletal approach [25] for parallel programming. There also exist similar methods for imperative languages such as the B-method [17], which models the abstract specification of a program to obtain a concrete C or Ada executable program.

Other approaches

There are tools that support different kinds of verification style. CompCert, for example, is a verified compiler for C programs [78, 74]. It has been proved that CompCert generates executable code that behaves exactly as described by the source program. Operating system architectures, called kernels, are subject to verification. The microkernel seL4 is an example of an operating-system kernel that has been proved correct [103]. The verification ensures that the system is free of implementation bugs (e.g., deadlocks or buffer overflows).

As mentioned previously, SYDPACC has been designed for *correctness-by-construction*. The provided functions of the algorithm allows for users to obtain correct parallel functions by construction. However, the construction of the framework is firstly made a posteri. Parts of this framework are built using already defined components to construct new terms, correct by construction. More details about how are constructed SYDPACC terms are given in Section 2.4 of Chapter 2, and in Chapter 3.

Contribution and organization

The contribution presented in this thesis is decomposed into two parts. First, we propose an extension of SyDPaCC, that already provides correct skeletons on lists, to cover parallel programs on trees. A skeleton is a notion introduced by Murray Cole [25]: it is a parallel implementation of a computation pattern. Using this approach, a developer does not have to think about parallelization anymore but only on how to write a program using these specific patterns. Using SyDPaCC, programmers are able to write correct parallel programs without taking care of parallel aspects. However, SyDPaCC is a Coq framework and Coq is not easy to use.

Beyond the importance of formal verification, easing the writing of parallel programs remains an important task. Then we propose PySke, a Python API that allows writing parallel programs on two different data structures. It covers parallel programming on lists but more importantly on trees, that are not often represented on already existing skeletons API. PySke takes advantage of Python to mainly increase programming productivity. It is indeed convenient to write skeletons in Python since lambda expressions are part of the language.

The thesis is organized as follows. Chapter 2 gives all the technical preliminaries needed to understand this document. It includes more details about BMF, Coq, BSML, and the construction of SyDPaCC for lists. In Chapter 3, we describe an extension of SyDPaCC for trees. Chapter 4 details PySke , a library of skeletons written in Python. Examples that can be written with skeletons are presented in Chapter 5. This Chapter also provide results of experiments on the presented examples on two parallel machines. We finally conclude and present future work in Chapter 6.

Preliminaries 2

Contents

In this chapter we present various conceptual and software tools to ease the reading of the contribution chapters. First, I begin to present in more details the Bird Merteens Formalism (Section 2.1). We describe here several techniques based on equivalences and theorems to optimize and automatically parallelize programs. Then functional parallel programming with Bulk Synchronous Parallel

ML (BSML) is introduced in Section 2.2. An overview of the Coq proof assistant – that I used to implement and prove the correctness of functions manipulating several representations of trees – is given in Section 2.3. This chapter ends with a presentation of SyDPaCC a library for Coq that supports reasoning on parallel functional programs, an in particular BSML programs, in a way that provides an automatic parallelization mechanism within Coq (Section 2.4). The contributions I describe in Chapter 3 are part of an extension to SyDPaCC.

Bird-Meertens Formalism (BMF)

The construction of categorical data types addressed software and performance issues in parallel programming [120, 68, 27]. The BMF describes a calculus for the construction of programs. It can be decomposed into several parts: notations, theorems, primitives, and transformations. The remaining of this section gives an overview of BMF on lists, and some-linear structures.

Notations

Type

To denote than an element a has type A, we write $a : A$. A tuple is defined with the composition of several types. For example, the type of a pair of an A element defined by a, and an element b of type B is given by $(a, b) : A \times B$.

Functions

The notation of functions is derived from lambda calculus and are defined with a functional style. Then the application of a function f to an element a is written $f.\ a$ or more simply $f\ a$ in functional languages. A function is also considered as an element of the formalization and then has a type. A function f which takes an argument of type A and returns an element of type B is typed $f : A \to B$. By convention, we use lambda notation to give the core of an anonymous function. A function that associates the input x to the expression e is written $\lambda x.e$. The ML syntax denotes the same function by $\lambda x \Rightarrow e$. For readability, we will be kept this one for the rest of the paper. We also introduce pattern matching for the parameters of functions. For example, a function that takes a tuple as argument will be written $\lambda(a, b) \Rightarrow e$, and typed $A \times B \to E$ with A (resp. B) the type of a (resp. b), and E the type of the returned expression.

To successively apply several functions to an entry, we use composition. The composition of two functions f and g is denoted by $f \circ g$ and is defined by $(f \circ g)\ a \ = \ f\ (g\ a)$. In functional programming, $f\ g\ a \ \neq \ f\ (g\ a)$. $f\ g\ a$ represents a function that takes as input two arguments: g and a. Considering $B \to C$ the type of g and A the type of a, f has the following type: $(B \to C) \to A \to E$, with E the type of the returned expression. In this case, $f\ g$ represents a partial application. $f\ g$ is now a function that takes as argument a variable of type A and returns an expression of type E. This technique is called currying. Uncurrying is its dual transformation. For example, the uncurried version of f defined previously would have the type $((B \to C) \times A) \to E$.

The tupling operation consists on creating a tuple from a set of functions. Each function will be applied to the same parameter and as a result we obtain a tuple of subresults: $f \triangle g = \lambda x \Rightarrow (f\ x,\ g\ x)$. The pairing operation works on a similar way but instead of having one input, we have an input for each function. Each function will be applied to its corresponding input. The tupling denoted by $\times$ is defined by $(f \times g)\ (x,\ y) = (f\ x,\ g\ y)$.

Usual functions

First of all, the *id* function defined by $id = \lambda x \Rightarrow x$. There are also functions to handle tuples, especially the pairs. To make a projection of the first element (resp. the second element) of a pair we use the function fst (resp. snd) defined by $fst(a, b) = a$ (resp. $snd(a, b) = b$).

Binary Operators

The BMF simplifies the use of binary operators by infix operators. By convention, they are written with $\oplus$, $\otimes$ or other similar symbols. $x \oplus y$ is equivalent to $(\oplus)\ x\ y$. More than simplify the notation of the operation, the infix operators allow an order in the application by sectioning the application. Indeed, the operators are not necessarily symmetric. That is, $x \oplus y$ is not necessarily equal to $y \oplus x$. Suppose that $\oplus : A \times B \to C$, we can define the 'left section' by $(a\oplus) : B \to C$ and the 'right section' by $(\oplus b) : A \to C$. Thus we have the equality: $(a\oplus)\ b = (\oplus b)\ a = a \oplus b$.

Inverse

The inverse of function f, denoted by f^{-1}, is a function satisfying $f \circ f^{-1} = f^{-1} \circ f = id$. The right inverse of f, denoted by f°, satisfies $f \circ f^{\circ} \circ f = f$. It is important to notice two things about right inverses. Firstly, there exists a right

inverse for any function. Worrying about its existence is then unnecessary [109]. Secondly, a right-inverse of a function is not unique. f° denotes one among the possible right inverses of f

Proof

A proof in BMF is constructed step by step using hypothesis and definitions as justifications on each step. The justifications are written in curly brackets, and a proof of equivalence between an original specification and a final version is as written below.

Proof. original specification = final version
original specification
= { *justification for the first step* }
transformed version
= { *justification* }
...
= { *justification for the final step* }
final version

□

The square marks the end of the proof.

Lists

A list is an homogeneous sequence of values of the same type. A list of elements of type A is defined by the type $List\ A$.

In the BMF tradition, join-lists are lists built using three constructor: the empty list $[\,]$, a singleton $[a]$, and the concatenation of two lists $xs +\!\!+ ys$. As we will explain in Section 2.1.3, join-lists are well-suited for parallelism.

In this formalization of lists, $+\!\!+$ is assumed to be an associative operation, and $[\,]$ is its left and right identity, or in short the set of lists, $+\!\!+$, and $[\,]$ form a monoid:

Definition 2.1.1 (Monoid) *A set A, a binary operation $\oplus$ on A, and a element $\iota_\oplus$ form an algebraic structure called a monoid if:*

- *$\oplus$ is associative:* $\forall x\ y\ z,\ (x \oplus y) \oplus z = x \oplus (y \oplus z)$,
- *$\iota_\oplus$ is both a left and right identity for $\oplus$:*

- $\forall x,\ \iota_{\oplus} \oplus x = x$
- $\forall x,\ x \oplus \iota_{\oplus} = x$

Having these properties mean that in this formalization there is no unique representation of a given list built using only constructors. In modern functional programming languages such a OCaml and Haskell, new data types can be defined by enumerating their constructors. In this context however, for a given list there is a unique representation of the list in terms of constructors. While it is possible to support formal reasoning in Coq following the BMF join-list style, it is much more convenient to adopt the functional programming perspective, in particular because ultimately the goal is to obtain functional parallel programs.

Therefore, we will rather use cons-lists, i.e. lists built using too constructors: $[\,]$ and :: (also named cons). In the join-list view, cons can be defined as: $x :: xs = [x] +\!\!+ xs$. In the cons-list view, $+\!\!+$ can be defined as follows:

$$[\,] +\!\!+ ys = ys \tag{2.1}$$
$$(x :: xs) +\!\!+ ys = x :: (xs +\!\!+ ys) \tag{2.2}$$

Lemma 2.1.2 *For all type A, List A, $+\!\!+$, and $[\,]$ form a monoid.*

Proof. Three properties must be proved: $[\,]$ is left neutral, $[\,]$ is right neutral and $+\!\!+$ is associative.

1. $[\,]$ is left neutral ($\forall\ ys,\ [\,] +\!\!+ ys = ys$):
   True by definition of $+\!\!+$
2. $[\,]$ is right neutral ($\forall\ ys,\ ys +\!\!+ [\,] = ys$):
   This proof is made by induction. In other words, we consider two cases:
   - Basic case: $ys = [\,]$
     $[\,] +\!\!+ [\,] = [\,]$ is true by definition.
   - Inductive case: $ys = x :: xs$
     We set the induction hypothesis: $xs +\!\!+ [\,] = xs$.
     $$\begin{aligned} & (x :: xs) +\!\!+ [\,] \\ = \ & \{\ \textit{definition of } +\!\!+, \text{ case (2.2)}\ \} \\ & x :: (xs +\!\!+ [\,]) \\ = \ & \{\ \textit{induction hypothesis}\ \} \\ & x :: xs \end{aligned}$$

3. $++$ is associative ($\forall\ xs\ ys\ zs,\ (xs ++ ys) ++ zs = xs ++ (ys ++ zs)$):

   This proof is made by induction. We have two possible cases for xs:

   - Basic case: $xs = [\,]$

     $$\begin{array}{ll} & ([\,] ++ ys) ++ zs \\ = & \{\ \textit{definition of } ++, \text{ case (2.2)}\ \} \\ & ys ++ zs \\ = & \{\ \textit{definition of } ++, \text{ case (2.1)}\ \} \\ & [\,] ++ (ys ++ zs) \end{array}$$

     We proved s$\forall\ xs\ ys\ zs,\ xs = [\,] \Rightarrow (xs ++ ys) ++ zs = xs ++ (ys ++ zs)$.

   - Inductive case: $xs = x_0 :: xs_0$

     We set the induction hypothesis: $(x_0 ++ ys) ++ zs = x_0 ++ (ys ++ zs)$.

     $$\begin{array}{ll} & ((x_0 :: xs_0) ++ ys) ++ zs \\ = & \{\ \textit{definition of } ++\ \} \\ & (x_0 :: (xs_0 ++ ys)) ++ zs \\ = & \{\ \textit{definition of } ++\ \} \\ & x_0 :: ((xs_0 ++ ys) ++ zs) \\ = & \{\ \textit{induction hypothesis}\ \} \\ & x_0 :: (xs_0 ++ (ys ++ zs)) \\ = & \{\ \textit{definition of } ++\ \} \\ & (x_0 :: xs_0) ++ (ys ++ zs) \end{array}$$

□

As already seen for function $++$, to write a function on lists, each case of construction must be considered. For example, a definition of *map* is:

$$\begin{cases} map\ f\ [\,] & = & [\,] \\ map\ f\ (x :: xs) & = & (f\ x) :: (map\ f\ xs) \end{cases}$$

map is a very common primitive on data structures. Another one is *reduce*. Considering an associative operation $\oplus$ and its neutral element $\iota_\oplus$, *reduce* can be informally defined as follow:

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

We introduce two other primitives called *scan* and its reverse version *rscan*. The scan operations on lists, *scan* and *rscan*, takes an associative operation $\oplus$

with its neutral element $\iota_\oplus$, and a list. Informally, these two functions are defined as follows:

$$\begin{aligned} scan\ (\oplus)\ [x_1;\ x_2;\ \ldots;\ x_n] &= [\iota_\oplus; \iota_\oplus \oplus x_1; \ldots; \iota_\oplus \oplus x_1 \ldots \oplus x_{n-1}] \\ rscan\ (\oplus)\ [x_1;\ x_2;\ \ldots;\ x_n] &= [x_2 \oplus \ldots \oplus x_n; x_3 \oplus \ldots \oplus x_n; \ldots; x_n; \iota_\oplus] \end{aligned}$$

The last common general primitive on lists is *map2* defined by:

$$map2\ f\ [x_1;\ x_2;\ \ldots;\ x_n][y_1;\ y_2;\ \ldots;\ y_n] = [f(x_1, y_1);\ f(x_2, y_2);\ \ldots;\ f(x_n, y_n)]$$

However, the more specific *zip* is often used on lists. It is just a particular case of *map2* where $f(x, y) = (x, y)$.

$$zip\ [x_1;\ x_2;\ \ldots;\ x_n][y_1;\ y_2;\ \ldots;\ y_n] = [(x_1, y_1);\ (x_2, y_2);\ \ldots;\ (x_n, y_n)]$$

Program transformations and parallelism

The most important thing about BMF is the program equivalences it provides [52, 6]. These equivalences, proposed as theorems, aims at facilitating parallel implementations of programs. In a naive and inefficient algorithm, specific classes of function can be detected and successively transformed into a more efficient combination of BMF terms. Using BMF and its primitives written as high-order functions allow generic definitions, both of the primitives and their parallel implementations. It makes possible the definition of a larger spectrum of programs.

Equivalences for optimizations

Equivalences provided by BMF aims at optimizing programs. For example, considering two functions f and g, it is more efficient to compute $map(f \circ g)$ than $(map\ f) \circ (map\ g)$. In the first case, we just go though the list once, while we do it twice in the second one.

Other examples show equivalences between compositions of different functions. Considering $\forall f\ g\ \oplus,\ (f \oplus \hat{}\, g)\ x = (f\ x) \oplus (g\ x)$, the following equality holds:

$$\begin{gathered} ((reduce\ (\oplus)) \circ (map\ f)) \oplus \hat{}\,((reduce\ (\oplus)) \circ (map\ g)) = \\ (reduce\ (\oplus)) \circ (map\ (f \oplus \hat{}\, g)) \end{gathered}$$

In this example, the right part of the equality uses two times less primitives and does not need a final use of $\oplus$. We can expect better performances for a program written using the right part than the left one.

Cases can be more context dependent and can take advantage of algebra. For instance, considering booleans by $true$ and $false$, and the functions not, or, and and, we have the following equivalence.

$$not \circ (reduce\ or) = (reduce\ and) \circ (map\ not)$$

In this case, the composition of *map* and *reduce* is obviously less efficient since we need to browse the list twice.

Theorems of equivalence

Finding specific patterns that can be transformed as compositions of BMF primitives is also very interesting for parallelization. Since these primitives can be implemented in parallel, programs can automatically be parallelized. In other words, if we can find an equivalence between the specification of a program and a composition of terms which have a parallel implementation, then this program be automatically parallelized. We present first an example of transformation following our instinct. Nonetheless, some theorems exist for automatic transformations such as the diffusion theorem and homomorphism theorems. The following paragraphs give details of these notions.

Example of composition: the MPS problem

An example of applications writable with BMF primitives is the Maximum Prefix Sum (MPS) problem. A prefix sum requires the binary associative operator (+), and a non-empty list l. Indeed, it doesn't make any sense to compute prefixes of an empty list, and even less a maximum among them. A sequential implementation to solve this problem can be written as follows.

$$\begin{array}{lll} mps\ l = mps'\ l\ (-\infty)\ 0 & & \\ \textbf{with} \quad mps'\ [\,]\ max\ acc & = & max \\ \qquad\quad mps'\ (x :: xs)\ max\ acc & = & mps'\ xs\ (max \uparrow (acc + x))\ (acc + x) \\ \textbf{and} \quad a \uparrow b = if\ a \geq b\ then\ a\ else\ b & & \end{array}$$

However, this program is not well-suited for parallelism. The computation of such a program can be split into three parts following the definition of the problem. First, all the possible prefixes must be listed. Secondly, for each prefix, the

sum of elements must be calculated. Finally, the program returns the maximum among the sums. We define one function for each step: *prefix* that calculates all the possible prefixes of a list, *sum* that computes the sum of elements of a list, and *maximum* to get the maximum value among a list. *sum* and *maximum* are functions that can be expressed using *reduce*. Indeed, $sum = reduce\ (+)\ 0$ and $maximum = reduce\ (\uparrow)\ (-\infty)$.

prefix is a recursive function that can be defined using *map* as follows.

$$\begin{array}{rcl} prefix\ [\,] & = & [[\,]] \\ prefix\ x :: xs & = & [\,] :: (map\ (cons\ x)\ (prefix\ xs)) \\ & & \textbf{with}\ cons\ x = \lambda\ xs \Rightarrow x :: xs \end{array}$$

We now have all the keys to writes a program that solves the MPS problem using BMF primitives.

$$mps = maximum \circ (map\ sum) \circ prefix$$

From this definition close to the informal functional specification, and using a distributed list with parallel implementations of the primitives, we obtain a parallel implementation of a program to solve the MPS problem.

It is interesting to compare the complexity of the two algorithms. The first one's is largely better than the second one's in a sequential view. However, the second one provides an easy automatic parallel implementation. With enough processors, and a large dataset, we can expect better performances with the parallel approach than the sequential one.

Diffusion Theorem

The diffusion theorem [70] states that a recursive function using an accumulative parameter can be transformed into a composition of the functions *map*, *reduce*, *scan*, and *zip*. A function h, with $\oplus$ and $\otimes$ two associative operators, defined with the following recursive function:

$$\begin{array}{rcl} h\ [\,]\ c & = & g\ c \\ h(x :: xs)\ c & = & p(x, c)\ \oplus\ (\ h\ xs\ (c \otimes q\ x)) \end{array}$$

can be transformed into:

$$\begin{array}{rcl} h\ xs\ c & = & (reduce\ (\oplus)\ (map\ p\ as))\ \oplus (g\ b) \\ & & \textbf{with}\ bs +\!\!+ [b] = map\ (c\otimes)\ (scan\ (\otimes)\ (map\ q\ xs)) \end{array}$$

Homomorphism Theorems

Definition 2.1.3 (Homomorphism) *For a binary operator $\odot$, a function h on lists is said $\odot$-homomorphic iff, for all x and y, the following equality holds: $h(x +\!\!+ y) = h\ x \odot h\ y$.*

For an associative operator $\odot$ and its unit $\iota_\odot$, an homomorphic h is written $h = hom\ (\odot)\ f$ and can be described by

$$\begin{array}{rcl} h\ [\,] & = & \iota_\odot \\ h\ [a] & = & f\ a \\ h\ (x +\!\!+ y) & = & h\ x \odot h\ y \end{array}$$

The functions *sum* and *maximum* presented in paragraph 2.1.3 can be expressed as homomorphisms as follows.

$$\begin{array}{rcl} sum\ (x +\!\!+ y) & = & sum\ x\ +\ sum\ y \\ \Rightarrow sum & = & hom\ (+)\ id \\ maximum\ (x +\!\!+ y) & = & maximum\ x\ \uparrow\ maximum\ y \\ \Rightarrow maximum & = & hom\ (\uparrow)\ id \end{array}$$

Definition 2.1.4 (Leftwards and rightwards functions) *A function f on lists is called a leftwards function if there exists a binary operator $\oplus$ such that*

$$\forall a\ x,\ f\ ([a] +\!\!+ x) = a \oplus (f\ x)$$

Symmetrically, a function g on lists is called a rightwards function if there exists $\otimes$ such that

$$\forall a\ x,\ g\ (x +\!\!+ [a]) = (g\ x) \otimes a$$

Theorem 2.1.5 (The first homomorphism theorem) *Every homomorphism can be written as the composition of map and reduce:*

$$\begin{array}{rcl} \forall h\ f\ \odot, & h = hom\ (\odot)\ f \\ \Rightarrow & h = (reduce(\odot)) \circ (map\ f) \end{array}$$

Theorem 2.1.6 (The second homomorphism theorem) *Every homomorphism is both a leftward and a rightward function. In other words, an homomorphism is a function that computes a list both from left to right or from right to left.*

These two theorems lead to the third homomorphism theorem [53].

Theorem 2.1.7 (The third homomorphism theorem) *A function h is a list homomorphism iff there exist two binary operators $\oplus$ and $\otimes$ such that:*

$$\begin{array}{rcl} h([\,]) & = & \iota_{\odot} \\ h([a] +\!\!+ x) & = & a \oplus (h\ x) \\ h(x +\!\!+ [a]) & = & (h\ x)\ \otimes a \end{array}$$

If we can compute an homomorphism in both leftward and rightward manners, then there exists a divide-and-conquer algorithm to evaluate the function. It results the following lemma [53, 109]:

Lemma 2.1.8 *For a given function h, and two binary operators $\oplus$ and $\otimes$, if the following equations hold for all a and x:*

$$\begin{array}{rcl} h([a] +\!\!+ x) & = & a \oplus (h\ x) \\ h(x +\!\!+ [a]) & = & (h\ x)\ \otimes a \end{array}$$

Then $h = hom\ (\odot)\ \phi$ where $\odot$ and ϕ are defined as follows for all a, x, and y.

$$\begin{array}{rcl} \phi\ a & = & h\ [a] \\ x \odot y & = & h\ (h^{\circ}\ x +\!\!+ h^{\circ}\ y) \end{array}$$

Homomorphism theorems have been shown useful in the development of parallel programs [10, 26, 27, 53, 68, 69].

If a candidate function h can be expressed both as a leftwards and a rightwards function, then h is an homomorphism (theorem 2.1.7). Thanks to the first homomorphism theorem (theorem 2.1.5), h can be written as a composition of *map* and *reduce*. Now, we can use parallel versions of these primitives on a parallel implementation of lists.

MPS revisited with homomorphism theorems The homomorphisms theorems and the procedure for optimization can be applied to the MPS problem. A naive composition of functions to parallelize a program solving the MPS problem has been previously given as follows.

$$mps = maximum \circ (map\ sum) \circ prefix$$

We have already defined *maximum* and *sum* as homomorphisms.

To apply the third homomorphism theorem, we need to show that *mps* can both be represented as a leftwards and a rightwards function.

The first sum of prefix being the value of the single first element, it is easy to find $\oplus$ such that *mps* is a $\oplus$-leftwards function.

$$a \oplus b = 0 \uparrow (a + b) \Rightarrow mps([a] +\!\!+ x) = 0 \uparrow (a + mps\ x)$$

The second step consists on finding $\otimes$ such that mps is a $\otimes$-rightwards function. There is no obvious solution for

$$mps(x +\!\!+ [a]) = (mps\ x) \otimes a$$

Keep the already accumulated sum looks necessary for defining $\otimes$. We notice that the paired function ms such that $ms = mps \triangle sum$ is both leftwards with $a \oplus_{ms} (x_m, x_s) = (0 \uparrow (a + x_m),\ a + x_s)$, and rightwards with $(x_m, x_s) \otimes_{ms} a = (x_m \uparrow (x_s + a),\ x_s + a$. The third homomorphism is then applicable to ms. From the lemma 2.1.8, ms, there exist $\odot$ and ϕ such that $ms = hom\ (\odot)\ \phi$ with

$$\begin{array}{rcl} \phi\ a & = & ms\ [a] \\ x \odot y & = & ms\ (ms^\circ\ x +\!\!+ ms^\circ\ y) \end{array}$$

We first define $\phi\ a$ by

$$\begin{array}{rcl} \phi\ a & = & (mps\ [a], sum\ [a]) \\ & = & (0 \uparrow (a + mps\ [\,]),\ a) \\ & = & (0 \uparrow a,\ a) \end{array}$$

To define $\odot$, we need to specify a right inverse of ms. As a solution: $ms^\circ\ (m, s) = [m; s - m]$.

Proof.

$$\begin{array}{rcl} \forall x,\quad ms(ms^\circ(ms\ x)) & = & ms(ms^\circ(mps\ x,\ sum\ x)) \\ & = & (mps \triangle sum)\ [mps\ x;\ (sum\ x) - (mps\ x)] \\ & = & (mps\ [mps\ x;\ (sum\ x) - (mps\ x)], \\ & & sum\ [mps\ x;\ (sum\ x) - (mps\ x)]) \\ & = & (mps\ x,\ sum\ x) = ms\ x \end{array}$$

□

We now can define $\odot$ for all x_m, x_s, y_m, and y_s.

$$\begin{array}{rcl} (x_m, x_s) \odot (y_m, y_s) & = & ms(ms^\circ\ (x_m, x_s) +\!\!+ ms^\circ\ (y_m, y_s)) \\ & = & ms\ [x_m; x_s - x_m; y_m; y_s - y_m] \\ & = & (mps\ [x_m; x_s - x_m; y_m; y_s - y_m], \\ & & sum\ [x_m; x_s - x_m; y_m; y_s - y_m]) \\ & = & (0 \uparrow x_m \uparrow (x_s + y_m),\ x_s + y_s) \end{array}$$

Using the thirst homomorphism theorem, we have:

$$ms = (reduce\ (\odot)) \circ (map\ f)$$

By deduction, and the definition of *ms*, we finally get an efficient version of *mps* by

$$mps = fst \circ (reduce\ (\odot)) \circ (map\ f)$$

Trees

Graph calculus is not defined in BMF. However, the calculus defined on lists has been extended to trees. Trees are particular kind of graphs, often used in representing structured data such as organigrams or XML documents. They are acyclic graphs satisfying the following properties:

1. There is exactly one vertex with no entering edges, called the root;
2. All the vertices that are not the root have exactly one entering edge;
3. There exists a path from to root to all the other vertices.

A technique to reduce a graph into a tree has been introduced in [133] but here we are only considering already-defined trees and more particularly binary trees.

Binary Trees

Binary trees are non-homogeneous and non-linear structures. There are two ways of building tree: $Leaf(a)$ represents a tree with only one leaf that contains a value a, and $Node(b, l, r)$ represents a node built using a value b and two binary trees l and r . The values at the leaves should have the same type α, while the values at the nodes should have the same type β. In this case, the type of the binary tree is denoted with $BTree\ \alpha\ \beta$.

As the functions on lists, functions on trees are defined considering each case of construction. The primitives are pretty similar. For example, the *map* function is defined by:

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

The *map* function on trees takes two functions: k_L, applied to the values at the leaves, and k_N, applied to the values at the nodes. The other traditional function,

reduce, can be defined in the same way on binary trees.

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

map and *reduce* are enough for solving easy problems. For instance, the size of a tree can be calculated as follows.

$$size \;=\; reduce\ (+)\ \circ\ map\ (\lambda x \Rightarrow 1)\ (\lambda x \Rightarrow 1)$$

The *scan* function on lists is derived into two accumulation functions: the upwards and downwards accumulations respectively defined by *uAcc* and *dAcc*.

uAcc is similar than *reduce* but has the particularity of preserving the tree structure as a result.

$$\begin{cases} uAcc\ k\ Leaf(a) & = & Leaf(a) \\ uAcc\ k\ Node(b,l,r) & = & Node(b',\ uAcc\ k\ l,\ uAcc\ k\ r) \\ & & \textbf{with}\ b' := reduce\ k\ Node(b,l,r) \end{cases}$$

dAcc takes three arguments in addition to the input tree:

- g_L proceeds the accumulation to the left children of a node;
- g_R proceeds the accumulation to the right children of a node;
- c the current accumulated value.

$$\begin{cases} dAcc\ g_L\ g_R\ c\ Leaf(a) & = & Leaf(c) \\ dAcc\ g_L\ g_R\ c\ Node(b,l,r) & = & Node(c,l',r') \\ & & \textbf{with}\ l' := dAcc\ g_L\ g_R\ g_L(c,b)\ l \\ & & \textbf{and}\ r' := dAcc\ g_L\ g_R\ g_R(c,b)\ r \end{cases}$$

For example, these accumulative primitives can be used for numbering tree elements with a prefix traversing order as follows.

$$\begin{aligned} prefix = \;& dAcc\ (\lambda(c,\ (bl,\ bs)) \Rightarrow c+1)\ (\lambda c,\ (bl,\ bs) \Rightarrow c+bl+1) \\ & \circ uAcc\ (\lambda((ll,\ ls),\ b,\ (rl,\ rs)) \Rightarrow (ls,\ ls+1+rs)) \\ & \circ map\ (\lambda x \Rightarrow (0,1))\ (\lambda x \Rightarrow x) \end{aligned}$$

Two other convenient functions, *root* and *setroot*, are used in binary tree calculation.

$$\begin{array}{lcl@{\quad\textbf{and}\quad}lcl} root\ Leaf(a) & = & a & setroot\ Leaf(a)\ v & = & Leaf(v) \\ root\ Node(b,l,r) & = & b & setroot\ Node(b,l,r)\ v & = & Node(v,l,r) \end{array}$$

$$
\begin{array}{rcl}
map\ f\ RTree(v,ch) & = & RTree(f(v),\ [map\ f\ t \mid t \in ch]) \\
reduce\ (\oplus)\ (\otimes)\ RTree(v,ch) & = & v\ \oplus \Sigma_{\oplus}[reduce\ (\oplus)\ (\otimes)\ t \mid t \in ch] \\
uAcc\ (\oplus)\ (\otimes)\ RTree(v,ch) & = & RTree(\ \ reduce\ (\oplus)\ (\otimes)\ RTree(v,ch), \\
 & & \qquad [uAcc\ (\oplus)\ (\otimes)\ t \mid t \in ch]) \\
dAcc\ (\oplus)\ c\ RTree(v,ch) & = & RTree(c,\ [dAcc\ (\oplus)\ (c \oplus v)\ t \mid t \in ch]) \\
lAcc\ (\oplus)\ RTree(v,ch) & = & \textbf{let}\ rs := rscan\ (\oplus)\ [root\ t \mid t \in ch] \\
 & & \textbf{in}\ \ RTree(\iota_{\oplus},\ [setroot\ (lAcc\ (\oplus)\ t_i)\ r_i \mid t \in \#ch]) \\
rAcc\ (\oplus)\ RTree(v,ch) & = & \textbf{let}\ rs := scan\ (\oplus)\ [root\ t \mid t \in ch] \\
 & & \textbf{in}\ \ RTree(\iota_{\oplus},\ [setroot\ (rAcc\ (\oplus)\ t_i)\ r_i \mid t \in \#ch])
\end{array}
$$

Figure 2.1 – *Primitives for RTree.*

Rose Trees

The nodes on real trees don't have necessarily two children. In this case, a tree with an arbitrary shape is called a rose tree [106]. They are used in many scientist domains such as statistics with Bayesian clustering [21]. Contrary to binary trees, there is only one way to create a rose tree, using the $RTree(v,ch)$ with v the value contained in the current node, and ch a list of $RTree$ corresponding to the children of the node. The functions $root$ and $setroot$ are also defined on rose trees.

$$
\begin{array}{rcl}
root\ RTree(v,ch) & = & v \\
setroot\ RTree(v,ch)\ v_1 & = & RTree(v_1,ch)
\end{array}
$$

The definitions of primitives on rose trees are defined in Figure 2.1.

However, it is not very convenient to manipulate a tree with an unknown shape. With binary trees, we know that each node has exactly two children. We define two functions for transforming trees: $r2b$ transforms a rose tree into a binary tree and is described by

$$
\begin{array}{rcl}
r2b\ t & = & r2b'\ t\ [\,] \\
r2b'\ RTree(a,ts)\ ss & = & Node(a, r2b''\ ts, r2b''\ ss) \\
r2b''\ [\,] & = & Leaf(\_) \\
r2b''\ t :: ts & = & r2b'\ t\ ts
\end{array}
$$

while $b2r$ does the inverse operation

$$
\begin{array}{rcl}
b2r\ t & = & head(b2r'\ t) \\
b2r'\ Node(l,b,r) & = & RTree(n, (b2r'\ l)) :: b2r'\ r \\
b2r'\ Leaf(a) & = & [\,]
\end{array}
$$

These two functions respect $b2r \circ r2b = id$. We show in the Section 4.3.4 how these transformations can be used to write skeletons on rose trees. An example of transformation is presented in Figure 2.2.

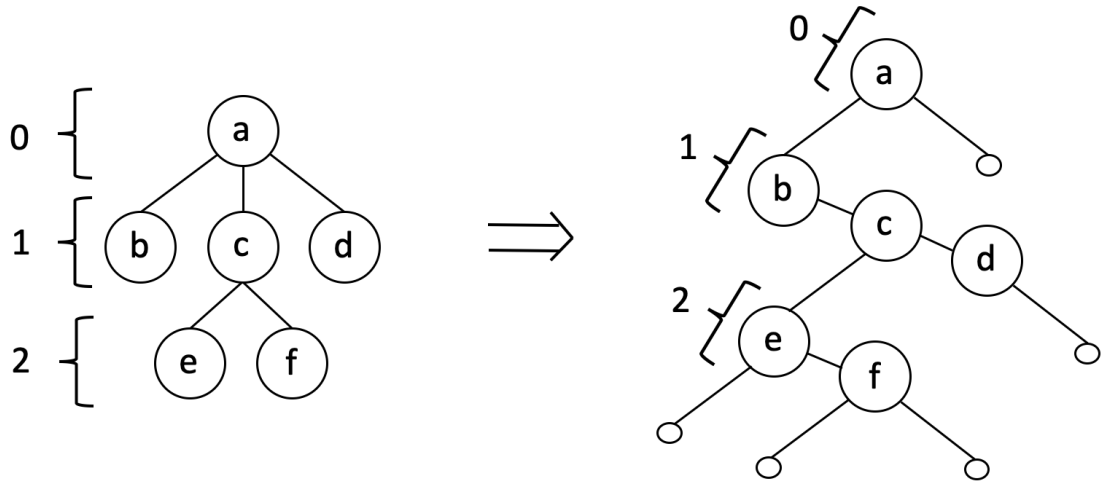


Figure 2.2 – *Example of RTree transformation into a BTree*

Diffusion on Trees

The diffusion theorem can be generalized to binary trees. If h is a function on binary trees defined with the following recursive way

$$\begin{array}{rcl} h\ Leaf(a)\ c & = & k_1(a,c) \\ h\ Node(b,l,r)\ c & = & k_2(b,c) \oplus (h\ l\ (c \otimes\ g_1\ b)) \oplus (h\ r\ (c \otimes\ g_2\ b)) \end{array}$$

then it can be transformed into

$$\begin{array}{rcll} h\ x\ c & = & (reduce\ (\oplus)\ (map\ k_1\ k_2\ ac)) & \\ & & \textbf{with} & \begin{array}{lcl} g_L\ c\ b & := & c \otimes (g_1\ b) \\ g_R\ c\ b & := & c \otimes (g_2\ b) \\ cs & := & dAcc\ g_L\ g_R\ c\ x \\ ac & := & zip\ x\ cs \end{array} \end{array}$$

Bulk Synchronous Parallel ML (BSML)

The Bulk Synchronous Parallel (BSP) model

The Bulk Synchronous Parallel (BSP) [104, 11] model is based on the PRAM model and was proposed by Valiant [131]. However, BSP does not take communication and synchronization for granted. Like MPI [49], BSP has explicit communication between processors.

Supersteps

Similarly to PRAM, the BSP algorithm computations proceed with a series of steps, called *supersteps*. Figure 2.3 presents an example of a BSP superstep. Each superstep is composed of three phases:

- **Computation**: a phase of asynchronous calculation during which every processors perform a local computation
- **Communication**: a phase of exchange. Each processor send and receive data to the others
- **Synchronization**: a barrier to synchronize all the processors before starting a new superstep.

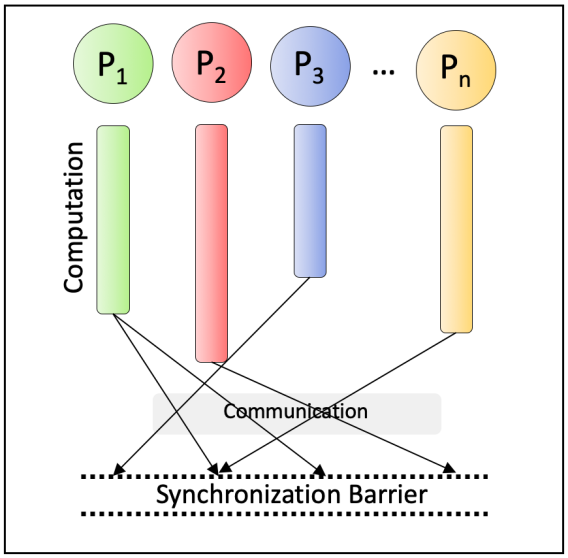


Figure 2.3 – *Example of a BSP superstep*

Bulk Synchronous parallel ML (BSML)

Language

The language BSML (Bulk Synchronous parallel ML) is a functional version of BSP based on the Ocaml (Objective Caml) language [128]. It has been designed as a library of OCaml [13, 93, 91]. BSML is a simple and structured language that allows us to write functional programs, with explicit parallelism on a quasi-synchronous machine. From the BSP approach, BSML programs are interblockage free, deterministic and performance estimable.

The main program in BSML handles a parallel vector structure with a fixed size `p` (the number of processors on a BSP machine), that contains, for each processor, a value. Such structure is represented by a polymorphic type `a' par` and will be denoted by $< x_0, ..., x_{p-1} >$ where x_i is the value contained in the i^{th} processor.

Primitives

BSML gives access to primitives but also BSP parameters. The number of processor `p` is accessible by `bsp_p`.

To get a value from a parallel vector, the primitive `get` takes as input a vector and a processor number and return the value it contains. Its signature is: `(α par → int → i` with `int` an integer value contained between 0 and `bsp_p - 1`.

The creation of a parallel vector is made with the primitive `mkpar : (int → α') → α par`. It takes a function `f` as a parameter and builds a parallel vector where for the processor `i`, the value is `(f i)`. Informally, `mkpar` is defined by:

```
mkpar f = <f 0, f 1, ..., f (p-1)>.
```

The parallel application of a vector that contains functions and a vector of value is possible thanks to the `apply` primitive. Its signature is:
`(α → β par → α par → β par`.

Using `mkpar` and `apply`, we can define `map` on parallel vectors.

```
# let map f = apply (mkpar (fun i -> f));;
val map : ('a -> 'b) -> 'a par -> 'b par = <fun>
```

`map` already exists in BSML referred as `parfun`.

The communications can be ensured with the primitive

```
let rec fold_left op e l =
  (* This function is equivalent to reduce on lists.
    It is also defined in the standard library of OCaml,
    can be used using List.fold_left *)
    match l with
    | [] -> e
    | x::xs -> fold_left op (op e x) xs;;
(* val fold_left : ('a -> 'b -> 'a)
        -> 'a -> 'b list -> 'a = <fun> *)

let reduce_step op e = mkpar (fun i -> fold_left op e);;
(* val reduce_step : ('a -> 'b -> 'a) -> 'a
        -> ('b list -> 'a) par = <fun> *)

let reduce op e v =
    (* Local reductions in the parallel vector *)
    let local = apply (reduce_step op e) v in
    (* List of all the processor ids *)
    let pids = (*  code omitted, eq. to [0;...; pid-1]*) in
    let results = List.map (proj local) pids in
    (* This last part cannot be parallelized.
    It makes a global reduction in sequential *)
    fold_left op e op e results;;
(* val reduce : ('a -> 'a -> 'a) -> 'a
                      -> 'a list par -> 'a = <fun> *)
```

Figure 2.4 – *Parallel implementation of reduce in BSML*

$$\texttt{proj} \;:\; \alpha\ \texttt{par} \rightarrow (\texttt{int} \rightarrow \alpha)$$

proj is the dual of **mkpar**. It takes a parallel vector and returns a function that associates a processor id to a value.

Using communications, we can now implement the BMF primitive *reduce* on parallel vectors. A definition of a parallel `reduce` is presented in Figure 2.4.

The Coq proof assistant

Proof assistants

A way to do a proof by hand is by using a composition of inference rules. An inference rule is presented as follows.

$$\textbf{rule}\frac{P_1 \quad \dots \quad P_n}{Q}$$

This expression states that if the premises $P_1 \quad \dots \quad P_n$ are respected, the consequence Q can be taken for granted. A rule without premises is called an axiom. For example, considering $s(n)$ the successor of a number (i.e. $s(n) = n + 1$), natural numbers can be defined by one axiom and one rule.

$$\textbf{zero}\frac{}{0\ \textbf{nat}} \qquad \textbf{succ}\frac{n\ \textbf{nat}}{s(n)\ \textbf{nat}}$$

Doing proof by hand is easy to write but easily wrong. A simple mistake in the proof process and the whole demonstration is false. Software have been developed to tackle errors and solve proofs. In the introduction, we mentioned Frama-C [119], a tool for verifications on sequential C programs. There are many proof assistants, and each has its characteristics. Some of them are based on the Curry-Howard correspondence, also known as the propositions-as-types paradigm. This isomorphism states that a proof $P \to Q$, where P and Q are propositions, is a function that takes as input a proof of P and returns a proof of Q. For an element a, a value of type Pa is a proof that Pa holds. It can be defined more visually with inference rules.

$$\frac{P \to Q \quad P}{Q} \equiv \frac{f : P \to Q \quad x : P}{f(x) : Q}$$

In this thesis, we use Coq [126]. Coq is a system developed in OCaml, based on the Curry-Howard correspondence. The system provides a language of tactics to help in proof solving. There also exist Isabelle [127]: a High Order Logic (HOL) prover, written in Standard ML; Idris [15] that is a pure functional language with dependent types intending to be general purpose programming language; Agda [14]: both a dependently typed functional language and a proof assistant based on the propositions-as-types paradigm; or also Rosette [129, 130], a solver-aided programming language extending Racket with language constructs for program synthesis and verification. This list is non-exhaustive.

Overview of Coq

The proof assistant Coq is based on the mathematical theory *CoC* (Calculus of Constructions). Coq is divided into three sublanguages:

- Gallina to write Coq terms (functions, types, axioms, etc.). Its syntax is very similar to OCaml's;
- Vernacular to control the behavior of the proof assistant;
- LTac, a language of tactics allowing to construct proofs interactively.

Every term of Gallina have a type, and the types are also terms of the language.

Type definitions

Every object of Coq has a type, including types themselves. The types are defining ordered sorts with `Set` as the bottom of the hierarchy. The following inclusions hold.

$$Set \leqslant Type_0 \leqslant Type_1 \leqslant Type_2 \leqslant ...$$

For any $i < j$, a $Type_i$ is typed by $Type_j$. In implies that the particular case of Set is typed by $Type_i$ with any i. Since `Set` is the type of the "small" datatypes and function types, cannot directly or indirectly involve other types [73].

As indicated on its original name, Coq is based on the calculus of construction theory. A definition is made using **`Definition`** and is constructed as follows.

`Definition` *name* : *type* := *definition*.

The pure type system from the CoC has been extended with inductive definitions from the Calculus of inductive Constructions (CiC). It is possible to write an inductive definition using the keyword **`Inductive`**. For example, natural numbers are defined in the standard library by:

```
Inductive nat : Set :=
    | O : nat
    | S : nat → nat.
```

The representation of numbers with `nat` is not efficient. That is why, the two sets `N` and `Z` have been defined, describing respectively natural numbers and

the whole set of integers (positive and negative). They are described with the inductive type positive, that is a representation of a binary number.

```
Inductive positive : Set :=
 | xI : positive → positive
 | xO : positive → positive
 | xH : positive.

Inductive N : Set :=
 | N0 : N
 | Npos : positive → N.

Inductive Z : Set :=
 | Z0 : Z
 | Zpos : positive → Z
 | Zneg : positive → Z.
```

Another type can parametrize the definition of a type. It is very convenient to use parameters for polymorphic structures. For example, the lists are defined in Coq by:

```
Inductive list (A:Type) : Type :=
    | nil : list A
    | cons : A → list A → list A.
```

In the case of `list` the polymorphism is explicit. The constructors of `list` will take a type as a first argument. However, it can be turned implicit by specifying to Coq the arguments that can be guessed by its type inference system in most situations.

```
Implicit Arguments nil [A].
Implicit Arguments cons [A].
```

The type inference system of Coq is strongly context dependent. The use of single `nil` is not enough for guessing the type A. `Definition l := nil` will return an error. To specify implicit arguments of a type, or a function, we need to use @. A working definition of an empty list is then: `Definition l := (@nil A).`

Coq provides notations to make more readable Coq code. For example, using the library ListNotations:

- Instead of `nil` and `cons h t`, we can respectively use `[ ]` and `h :: t`;
- Singletons can be defined by `[x]`;
- The list containing `x1`, `x2` and `x3` can be written `[x1; x2; x3]`
- The concatenation of two lists `a` and `b` can be made using an infix operator: `a ++ b`.

A very useful type defined in the standard library is `option`. `option A` is an extension of `A` with an extra element `None`.

```
Inductive option (A:Type) :=
  None : option A
| Some : A → option A.
```

Functions

Coq provides a pattern matching mechanism for defining functions. By filtering cases characterized by a pattern, different behaviors can be defined. However, it is important to notice that all the functions in Coq must be total. In other words, a function must determine behavior for every constructor of inputs.

For example, the functions *pred* that returns the predecessor of a natural number can be defined by:

```
Definition pred (n: nat) : nat :=
match n with
| O ⇒ O
| S m ⇒ m
end.
```

Besides, to set a recursive function, the keyword **`Definition`** must be replaced by **`Fixpoint`**. The *map* function on lists can be described as follows.

```
Fixpoint map (A B:Type) (f: A → B) (l: list A) : list B :=
match l with
| [ ] ⇒ [ ]
| h :: t ⇒ (f h) :: (map A B f t)
end.
```

Coq only allows definitions of functions that terminate. A recursive function must have a decreasing argument. using the implicit argument {struct}. For example, if the decreasing argument is a list l, the function can take a argument {struct l}. Another way to define a decreasing argument with a measure is defined in the section 2.3.2.

A value can be specified with a property using the set sig. It is composed of two parts: a value, and a proposition on this value. (sig A P), or by using a more suggestive notation, {x:A | P x}, denotes the elements of the type A which satisfy the predicate P. sig2 represents the same but allows to write two predicates: (sig2 A P Q), or {x:A | P x & Q x}, denotes the elements of the type A which satisfy the predicates P and Q. The value and the predicate(s) can be respectively got using the functions proj1_sig and proj2_sig.

Proofs in Coq

From definitions, it is possible to define lemmas, properties or theorems in Coq with related proof. For example, from the function length on lists that calculates the number of elements in the structure, and map defined previously, we define the following property.

```
Lemma map_length : ∀ A B (f : A → B) (l: list A),
length (map f l) = length l.
```

A proof of this lemma can be defined using LTac, the tactic language.

```
Proof.
 intros A B f l.
 induction l as [| x xs Hx].
 + simpl. reflexivity.
 + simpl. rewrite Hx; reflexivity.
Qed.
```

Let's analyze this step by step. First, we start the proof using the keyword Proof. The environment of Coq returns a response indicating that there is still one subgoal to prove.

```
1 subgoal
============================
∀ (A B : Type) (f : A → B) (l : list A), length (map A B f l) = length l
```

To start the proof, we need to introduce the variables we will use:
`intros A B f l.`

```
A : Type
B : Type
f : A → B
l : list A
============================
length (map A B f l) = length l
```

In our lemma, l is a list, and then its definition is made by induction. We need to do an induction on the structure of l for solving this proof:
`induction l as [ | x xs Hx]`. This tactic can be understood such there are two cases separated by |. The first case is the situation of l is `nil`. There is nothing to define here. Otherwise, l is `cons x xs`, and we name the inductive hypothesis with Hx. The answer of Coq shows two subgoals: one for each possible constructor of l.

```
A : Type
B : Type
f : A → B
============================
length (map A B f []) = length []

subgoal 2 is:
 length (map A B f (x :: xs)) = length (x :: xs)
```

By default, Coq makes us resolve the first one. Each subgoal resolving is marked by a bullet +. When the first one is started, Coq only keeps what can be used in the proof.

```
A : Type
B : Type
f : A → B
============================
length (map A B f []) = length []
```

According to the definition of length, length [] = 0. Since map A B f [] = [], length (map A B f []) can be simplified by 0 with the tactic `simpl`. The two expressions will be simplified, and the environment returns:

```
A : Type
B : Type
f : A → B
============================
0 = 0
```

Because equality is reflexive, the resolution can be finished by `reflexivity`. There is still the second subgoal to solve.

```
A : Type
B : Type
f : A → B
x : A
xs : list A
Hx : length (map A B f xs) = length xs
============================
length (map A B f (x :: xs)) = length (x :: xs)
```

Using the simplification with the tactic `simpl` the Coq response is the following.

```
A : Type
B : Type
f : A → B
x : A
xs : list A
Hx : length (map A B f xs) = length xs
============================
S (length (map A B f xs)) = S (length xs)
```

The resolution can be finished by using the inductive hypothesis and `reflexivity`. We process these two operations using a semi-column by `rewrite` → Hx; `reflexivity`.

Program definition

`Program Fixpoint` defines any recursive function and generates relative proof obligation. For example, the termination of a function can also be described using a measure of the arguments with the keyword measure. If there is no explicit

decreasing argument in the definition of a recursive function, a proof obligation about the decreasing measure is generated.

```
Program Fixpoint foo (A B : Type)  (l1 l2 : list A) {measure (length l1 + length l2)}
: list B :=
 match l1, l2 with
 | [], [] ⇒ []
 | h::t, [] ⇒ foo t []
 | [], h::t ⇒ foo [] t
 | h1::t1, h2::t2 ⇒ foo t1 t2
 end.
```

The obligations can be solved one by one using `Next Obligation`. However, Coq will try to solve them by itself. If it cannot, it will display them as remaining obligations to prove. Here, there is only one obligation that still needs to be solved.

```
A : Type
B : Type
h2 : A
t2 : list A
h1 : A
t1 : list A
foo : ∀ (A0 B : Type) (l1 l2 : list A0),
   length l1 + length l2 < length (h1 :: t1) + length (h2 :: t2) → list B
============================
length t1 + length t2 < length (h1 :: t1) + length (h2 :: t2)
```

Coq provides a tactic `omega` from the library Omega that can solve an equation of natural numbers automatically. This obligation can be resolved by simplification and the tactic `omega`.

```
Next Obligation.
    simpl; omega.
Defined.
```

Type classes and Instances

In mathematics, a structure is always accompanied by its properties. These properties are implicit and are not carried every time they are used. For instance,

the simple calculation $x + y$ involves the operation $(+)$ that is supposed defined with all its properties. In this case, I intentionally not defined a type for x and y. The operation won't be the same for integers or strings. Here, in both cases, a property could be that $(+)$ has a neutral element (0 or an empty string).

Type classes [122] in Coq is a solution for surcharging operators and specify abstract structures. They first have been introduced in Haskell, for ad-hoc polymorphism [132]. For example, it is not possible to define equality of predicates on natural numbers (`nat → nat → bool`), but it is possible to compare much types (boolean, integers, lists, etc.). Using type information of comparable, we want to compare two elements without surcharging notations. It is possible, with type classes in Coq, to define an equality definition using a class definition. The type class for equality is an override method =?.

```
Class Eq A :=
 {
  eqb: A → A → bool;
 }.

Notation "x =? y" := (eqb x y) (at level 70).
```

The keyword level in notations denotes a precedence level ranging from 0 to 100. Once a class is defined, we can determine its instances.

```
Instance eqBool : Eq bool :=
 {
  eqb := fun (a b : bool) ⇒
   match a, b with
    | true, true ⇒ true
    | false, false ⇒ true
    | _, _ ⇒ false
   end
 }.

Instance eqNat : Eq nat :=
 {
  eqb := Nat.eqb
 }.
```

Now, the operator =? is defined for `bool` and `nat` types. It is possible to

obligate a parameter of a function to be an instance of a class. The test of the difference between elements can be defined as follows.

```
Definition areDiff (A:Type) (a b: A) {H: Eq A} :=
 if a =? b then false else true.
```

The advantage of passing the hypothesis as an implicit argument is that the Coq system will automatically look for an instance of the type class in its definition, corresponding to the parameters. The resolution of instances is made with a backward manner, i.e. the last defined instance will be preferred during the resolution. It is important to notice that it is possible to sort the instances for the resolution. The order is defined by specifying a level value. The instance with the smallest value will be used, even if it has been defined before. 0 is the default value for an instance. In the example of Figure 2.5, the resolution will choose instance xor, and not or. If the levels was not specified, the instance or would be preferred.

Section

Coq code can be decomposed into sections. All the variables and hypothesis within a section are only declared for the section scope. For instance, the decidability of a list of A elements can be defined as follows.

```
Section EqDec.
 Variable A : Type.
 Hypothesis eq_a : ∀ (a1 a2 : A), {a1 = a2} + {a1 <> a2}.

 Lemma eq_dec_list : ∀ (l1 l2 : list A),
    {l1 = l2} + {l1 <> l2}.
 Proof.
  intros u v.
  repeat decide equality.
 Qed.
End EqDec.
```

Modules

Coq has a strong abstraction mechanism thanks to module types. In a module type, behavior is defined using a set of **Parameter**. A module is a particular definition of a module type.

```
Class Or A :=
 {
  orb: A → A → bool;
 }.

Instance xor : Or bool | 1 :=
 {
  orb := fun (a b : bool) ⇒
       match a, b with
       | true, false ⇒ true
       | false, true ⇒ true
       | _, _ ⇒ false
       end
 }.


Instance or : Or bool | 2 :=
 {
  orb := fun (a b : bool) ⇒
       match a, b with
       | true, _ ⇒ true
       | false, x ⇒ x
       end
 }.

Definition f_or (a b : bool) {H: Or bool} :=
 orb a b.
```

Figure 2.5 – *Examples of an instance resolution*

```
Module Type Numbers.
 Parameter A:Type.
 Parameter add: A → A → A.
 Parameter minus: A → A → A.
End Numbers.

Module Integers <: Numbers.
 Definition add := Nat.add.
 Definition minus := Nat.sub.
End Numbers.
```

A module type given as a parameter of another one gives accesses to its parameters, even if they are not explicitly implemented. Modules can be instantiated by providing a concrete implementation of the parameter model types.

```
Module Make (Import Numbers : Numbers).
 Definition add_minus x y := Numbers.minus y (Numbers.add x y).
End Make.

Module Import OpIntegers := Make Integers.
```

Coq modules can contained sections, but not the inverse.

Extraction

Coq has not been designed to compute efficiently. To tackle the lack of performance, it is possible to extract Coq code using the library Extraction [85] with three possible targeted languages: OCaml, Haskell, Scheme. These languages are only computing languages. Then, during the extraction, all the logical aspects are removed. That is, the extraction mechanism keeps only functions and definitions of types.

As an example, the function map defined page 33 and its dependencies can be extracted as the OCaml code shown in Figure 2.6

Finally, the command Separate Extraction get the extracted result into ml files.

```
type α list =
| Nil
| Cons of α * α list

(** val map : (α1 → α2) → α1 list → α2 list **)

let rec map f = function
| Nil → Nil
| Cons (h, t) → Cons ((f h), (map f t))
```

Figure 2.6 – *Example of extracted OCaml code from a Coq definition of the map function*

SyDPaCC

BSML in Coq with SyDPaCC

SyDPaCC is a set of libraries for Coq. It includes transformation theorems that state the equivalence of sequential functions and that are used to obtain efficient sequential programs from inefficient ones; BSML formal semantics; and a set of algorithmic skeletons, programmed using BSML and proved correct with respect to sequential functions. For example, SyDPaCC provides the first homomorphism theorem on lists that states that a homomorphic function is equivalent to a composition of *map* and *reduce*. We refer to [94, 39, 92, 90] for details about the existing theorems in SyDPaCC.

SyDPaCC defines two **Module** Type to formalize BSML: BSP_PARAMETERS that gives a definition of the parameters of the BSP machine, and BSML that defines the primitives of BSML. BSP parameters are defined with:

```
Parameter p : N.
Axiom p_spec : 0 < p.
```

The parameters of BSML are the signature of the primitive signature:

```
Notation pid := { n:N | N.ltb n Bsp.p = true }. (* N.ltb a b → a ≤ b *)
par : Type → Type.
Parameter get:
 ∀ (A:Type), par A → pid → A.
Parameter mkpar:
 ∀ (A:Type) (f:pid → A), par A.
Parameter apply:
 ∀ {A B:Type)(vf: par(A→B))(vx:par A), par B.
Parameter put:
```

```
 ∀ (A:Type)(vf:par(pid→A)),
   par(pid→A).
Parameter proj:
 ∀ (A:Type)(v: par A), pid → A.}
```

They are all accompanied by axioms to defined their semantics.

What is specific to BSML in SyDPaCC is the implementation of some algorithmic skeletons in BSML and the proof of their correctness with respect to some sequential functions. For example, parallel reduction is defined as the preduce skeleton:

$$\begin{aligned} &\mathit{preduce} \oplus\ v = \mathit{reduce} \oplus\ \mathit{list} \\ &\quad \text{where } \mathit{list} = \mathit{map}\ (\text{proj}\ \mathit{local})\ \mathit{pids} \\ &\quad\ \ \text{and } \mathit{local} = \text{parfun}\ (\mathit{reduce} \oplus)\ v \end{aligned}$$

In the definition of parfun, v has type **par**(`list` A); *reduce* is a sequential function; and *pids* is the list of all processor identifiers. parfun is function taking as argument a sequential function f and returning `apply`(**mkpar**(`fun` *pid*⇒f)). parfun has the same semantic that the one defined in the paragraph 2.2.2. $\oplus$ should be an associative function with a neutral element $i_{\oplus}$. Note that the return type of *preduce* is not a parallel vector but a value of type A.

The way the provided algorithmic skeletons are proved correct is the basis of the automatic parallelization feature of SyDPaCC that replaces sequential data structure types by parallel ones, and sequential functions by equivalent parallel ones. This mechanism relies on two notions of correspondence: type correspondence and function correspondence, both defined as type classes.

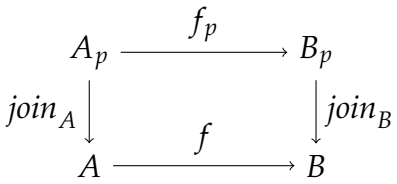
Type and function correspondences

The type correspondence between **par**(`list` A) and `list` A is based on a function join that reduces in parallel a parallel vector of lists into a list, the binary operator being list concatenation:
join v = preduce (++) (map (**proj** v) pids). An instance reduce_preduce expresses that preduce is correct with respect to reduce using the above type correspondence.

A sequential type seq_t corresponds to a parallel type par_t if there exists a join function that transforms any value of the parallel type to a value of the sequential type. To avoid too simple transformations (for e.g. a constant function that returns only one sequential value), join is required to be surjective. With this additional condition, any sequential value must be reached by join.

A parallel function fp:Ap→Bp is correct with respect to a sequential function f:A→B if there are type correspondences between A and Ap, and between B and

Bp. Additionally, f and fp should compute the same result: the result obtained by sequentializing the result of the application of fp to a parallel value should be the same that the result obtained by applying f to the sequentialization of the same parallel value. These requirements mean the follow diagram should commute:

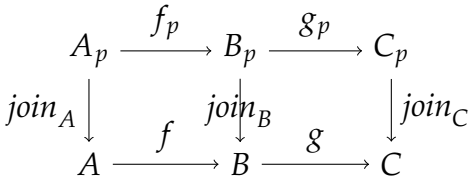


The class funCorr represents the correspondence between two functions. Similarly, TypeCorr represents the correspondence between two types. In both cases, joining functions are used to define how a type, or a function, is associated to another.

```
Class TypeCorr (seq_t:Type) (par_t:Type) (join:par_t→seq_t) :=
 { type_corr :> Surjective join }.

Class FunCorr
    ‘{ACorr : TypeCorr A Ap join_A } ‘{BCorr : TypeCorr B Bp join_B}
    ‘(f:A→B)  (fp:Ap→Bp) ‘{HP:@ParProp A Ap join_A ACorr P}:=
 { fun_corr : ∀ ap, P ap → join_B (fp ap) =  f (join_A ap) }.
```

One important property of function correspondence is that it can be easily composed. If fp is correct with respect to f and gp is correct with respect to g then fp ∘ gp is correct with respect to f ∘ g.



In a Prolog-like syntax (omitting the type correspondences), we have the following rule:

```
funCorr( f ∘ g, fp ∘ gp ) : funCorr(f, fp), funCorr(g, gp).
```

A Coq instance captures this property.

```
(*
 EnsuresProp is a class ensuring that all the results of a
 function respects a property
```

```
*)
Class EnsuresProp `(f:A→B)`(P:B→Prop) :=
 { ensures_prop: ∀ a:A, P(f a) }.

Instance fc_comp_fcensures
      `{ACorr : TypeCorr A Ap join_A} `{BCorr : TypeCorr B Bp join_B}
      `{CCorr : TypeCorr C Cp join_C}
      `{fCorr : @FunCorr A Ap join_A ACorr B Bp join_B BCorr f fp Pa HPa}
      `{gCorr : @FunCorr B Bp join_B BCorr C Cp join_C CCorr g gp Pb HPb}
      `{H: @EnsuresProp Ap Bp fp Pb }
 : FunCorr (g ∘ f) (gp ∘ fp).
```

Automatic parallelization

The automatic parallelization is provided by a function parallel that takes as argument a sequential function f. All its other arguments are implicit and include type and function correspondences. In particular an implicit argument is a function fp correct with respect to f. parallel just returns fp.

```
Definition parallel `(f:A→B)
       `{ACorr : TypeCorr A Ap join_A} `{BCorr : TypeCorr B Bp join_B}
       `{fCorr : @FunCorr A Ap join_A ACorr B Bp join_B BCorr f fp (fun x⇒True) HP} :
 Ap → Bp := fp.
```

This definition seems quite useless, but because of the implicit arguments, it is not. When applied to a sequential function, the search by Coq for an instance, will try to build a parallel function equivalent to f using instances of the type class FunCorr. Due to the instance about the composition of two correspondences, this search possibly includes decomposition of f into a composition of other sequential functions, and a search of equivalent skeletons for these functions. However, the Coq code of parallel programs is not made for being executed. But, thanks to the axiomatization of BSML in SyDPaCC, the extraction of code directly provides BSML programs that are, in addition, proved correct.

Lists in SyDPaCC

To illustrate how works SyDPaCC, we will consider the example of lists already defined in the framework.

BMF in SyDPaCC

Several aspects of BMF, such as notations, have been added in SyDPaCC. For example, tupling, pairing, and composition can be respectively written using the symbols $\times$, $\triangle$, and $\circ$. The operators also have algebraic properties. The class Monoid is a class representing an associative operator and its neutral element. Properties on operators such as associativity, commutativity and are also defined as classes. These definitions are given in Figure 2.7.

Sequential functions

All the sequential functions are presented in their simple form. However, they are all implemented as tail-recursive functions in SyDPaCC. For instance, the function map on lists is already defined in Coq, but is redefined as a tail-recursive function, map', in SyDPaCC. To ease the read of the functions, we will give the definitions with the names of functions that are not necessarily tail-recursive even if they are implemented as tail-recursive functions.

The function reduce is defined from fold_left, a left-to-right iterator on lists defined in the standard library of the language. fold_left2 is an alternative fold_left that takes two lists as input.

```
Fixpoint fold_left (l:list B) (a0:A) : A :=
 match l with
    | nil ⇒ a0
    | cons b t ⇒ fold_left t (f a0 b)
 end.

Fixpoint fold_left2 A B C (op:C→A∗B→C) (e:C) (l1:list A) (l2:list B): C :=
 match (l1, l2) with
 | ([], _ ) | (_ , []) ⇒ e
 | (h1::t1, h2::t2) ⇒ fold_left2 op (op e (h1,h2)) t1 t2
 end.

Definition reduce A (op:A → A → A) `{Monoid A op e} :=
 fun l ⇒ fold_left op l e.
```

The last primitive from BMF is the function *scan*. The left accumulation of values within a list is processed using an operator and its neutral element, represented by an instance of Monoid. In the definition of *scan*, the last element of the input list is not used during the calculation. It is however convenient in some

```
Class LeftNeutral A B (op: B → A → A) (e : B) :=
 {
  left_neutral  : ∀ a, op e a = a
 }.

Class RightNeutral A B (op: A → B → A) (e : B) :=
 {
  right_neutral : ∀ a, op a e =  a
 }.

Class Neutral A (op: A → A → A) (e : A) :=
 {
  neutral_left_neutral  :>  LeftNeutral  op e;
  neutral_right_neutral :>  RightNeutral op e
 }.

Class  Associative A (op:A→A→A) :=
 {
  associative : ∀ (x y z: A), op (op x y) z = op x (op y z)
 }.

Class Commutative A (op:A→A→A) :=
 {
  commutative : ∀ (x y: A), op  x y = op y x
 }.

Class Monoid A (op : A→A→A) (e : A) :=
 {
  monoid_assoc   :> Associative op;
  monoid_neutral :> Neutral op e
 }.
```

Figure 2.7 – *Definition of classes in* SyDPaCC

cases to accumulate it anyway. The function scanl_last is a version of scan that keeps this accumulation.

Finally, another function is added to the set of BMF functions. *accumulate* is formally defined and implemented as follows.

$$\begin{cases} accumulate\ g\ p\ q\ \oplus \otimes\ c\ [] = g\ c \\ accumulate\ g\ p\ q\ \oplus \otimes\ c\ (x :: xs) = \\ \quad p(x, c) \oplus (accumulate\ g\ p\ q\ \oplus \otimes\ c\ xs) \end{cases}$$

```
Fixpoint accumulate (A B C:Type) (g:B→C) (p:A*B→C) (q:A→B)
    (oplus:C→C→C) (otimes:B→B→B) (c:B) (l:list A) : C :=
 match l with
 | [ ] ⇒ g c
 | x::xs ⇒ oplus (p (x,c)) (accumulate g p q oplus otimes (otimes c (q x)) xs)
 end.
```

List skeletons

The skeletons for lists are defined for distributed lists with an SPMD approach. In other words, SyDPaCC manipulates vectors of lists, each representing a subpart of a global list.

By convention, the skeletons in SyDPaCC are defined by the names par_fct with fct one of the sequential function described above. For example, par_map, par_reduce, and par_scanl_last are defined as follows.

```
Definition par_map (A B:Type)(f:A→B) : par(list A)→par(list B) :=
  parfun (map f).

Definition par_reduce {A}(op:A→A→A){e:A}{H:Monoid op e}(v:par(list A)) : A :=
  let local := parfun (reduce op) v in      (* local reductions *)
  let list  := List.map (proj local) pids in (* vector → list  *)
  reduce op list.       (* reduction of the partial reductions *)

Definition par_scanl_last A B op1 op2 (e:B) c (v:par(list A)) : par(list B) * B :=
  let partials := parfun (scanl_last' op1 e) v in
  let scanls   := parfun fst partials in
  let lasts    := parfun snd partials in
  let sums     := par_scanl1 op2 c lasts in
  ( parfun2 (fun sum⇒map (op2 sum)) sums scanls,
    par_last (parfun2 op2 sums lasts) ).
```

In these definitions, parfun2 is an alternative version of parfun that takes two parallel vectors and is defined as follows.

```
Definition parfun2 A B C (f:A→B→C)(v:par A)(v':par B) : par C :=
  apply (parfun f v) v'.
```

There is no par_scan skeleton in SyDPaCC. Indeed, scan is just a particular case of scanl_last.

Correspondences and Automatic Parallelization

As explained in the introduction, the parallelization of programs is automatic if some conditions are respected. First, the distributed type used in skeletons must have a sequential correspondence. Skeletons on lists are defined on distributed lists. Then, we need a surjective function join that transforms a parallel vector of lists into a single list. To get this single list, we need first to get all sublists by processors and concatenate them together.

```
Program Definition join A (v:par(list A)) :=
    List.flat_map (proj v) pids.

Global Program Instance list_par_list_corr A :
    TypeCorr (@join A).
```

We notice here the presence of `Global Program` before the definition of the instance. It can be view as two parts. First, `Global` extends the effect outside the current sections and current module. `Program` then uses the obligation mechanism to manage missing fields, as described in section 2.3.2.

Now the correspondence between functions must be established for the pattern we want to make parallel. For example, for map and reduce:

```
Global Program Instance  map_par_map A B (f:A→B) : FunCorr (map f) (par_map f).

Global Program Instance reduce_par_reduce '(op:A→A→A) '{Monoid A op e} :
  FunCorr (reduce op) (par_reduce op).
```

Thanks to diffusion theorems, it has been stated that *accumulate* can be rewritten using other primitives as follows.

$$\left\{ \begin{array}{ll} & accumulate\ g\ p\ q\ \oplus \otimes\ c \\ = & (prod\_curry \oplus) \circ (prod\_curry\ (fold\_left2 \odot \iota_{\oplus}) \times g) \circ \\ & ((id \times fst) \triangle (snd \circ snd)) \circ (id \triangle (scanl\_last \ominus c)) \end{array} \right.$$

In this equality, *prod_curry f* when f has type $A \to B \to C$ is the same function than f, but as type $A \times B \to C$. Since the composition of function correspondences is defined in SyDPaCC, accumulate is automatically parallelized with all the function correspondences defined in the framework.

Correct parallel patterns for trees

3

Contents

In this chapter, we present an extension of the Coq framework, SyDPaCC, with skeletons for binary trees. Binary trees are more complex than a classic linear data structure such as lists: the tree structure is irregular, and not necessarily balanced. Matsuzaki et al. have designed parallel skeletons [75, 99] for manipulating binary trees. We designed a functional formulation of these algorithms and we implemented algorithmic skeletons on trees using a formalization of BSML in the Coq proof assistant. We also stated the correctness of these skeletons, and proved it for some. The full code of the extension is available at `https://www.dropbox.`

com/s/l5echieu9wjcm6q/sydpacc_defense.tar.gz. We first present the different representations we used to define trees (Section 3.1). Then we present the implementation of functions for these particular datatypes (Section 3.2) and their parallel implementation (Section 3.3) with the corresponding equivalences (Section 3.4). We discuss related work in Section 3.5.

Types and representations

Binary trees as an inductive type

The definition of types is following the one presented in Section 2.1.4. In Coq, it is given as follows.

```
Inductive t (A B: Type) :=
| Leaf : A → t A B
| Node : B → t A B → t A B → t A B.
```

Since this type is defined in a library named BTree, it is named BTree.t.

Serialized trees

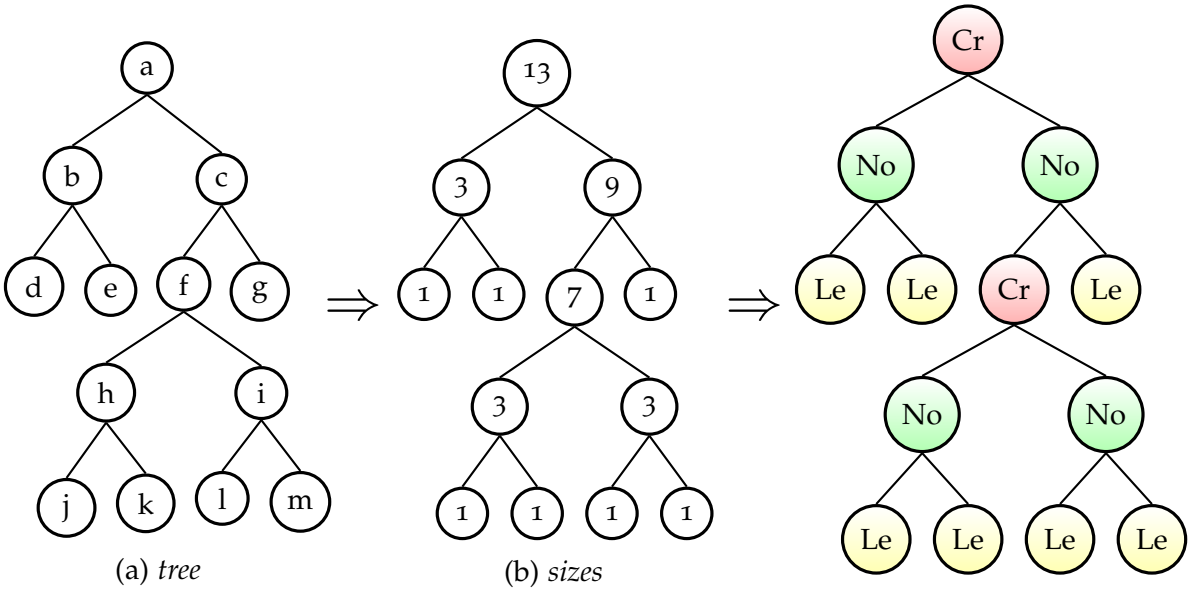
Since a node of a binary tree has two children, the distribution of the data can be discussed: The tree has to be cut into segments, but how? Here, the structure is linearized and divided as follows. Given an integer m, a node v is called m-critical if, for each v' child of v, the following inequality is respected: $\lceil size(v)/m \rceil > \lceil size(v')/m \rceil$ with *size* defined by:

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

The critical nodes are the cut points of the tree. Each subtree is translated into a list of value, defined by the inductive type value. The constructors describe the elements in a binary representation: Le denotes a leaf, No denotes a normal node and Cr a critical node. A segment is then a list of values, and a linearized tree, a list of segments.

```
Inductive value A B :=
 | Le : A → value A B
 | No : B → value A B
 | Cr : B → value A B.

Definition segment A B := list (value A B).
```



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

```
(* Considering A (resp. B) as the type of leaf (resp. node)
   values *)
Definition s1 : segment A B := [(Cr a)].
Definition s2 : segment A B := [(No b); (Le d); (Le e)].
Definition s3 : segment A B := [(No c); (Cr f); (Le g)].
Definition s4 : segment A B := [(No h); (Le j); (Le k)].
Definition s5 : segment A B := [(No i); (Le l); (Le m)].
Definition lt : list (segment A B) := [s1; s2; s3; s4; s5]
```

Figure 3.1 – *Example of list representation of a binary tree (with $m = 5$)*

The serialization function proceeds in several step. First, the size of each subtree is calculated in order to annotate the tree elements by Cr, No, or Le. The tree is cut on the critical nodes and each subtree is transformed into a list. Figure 3.1 gives an example of a serialized tree with $m = 5$.

The reverse operation, deserialization, proceeds in several steps. First, each segment is transformed into a subtree (segments_to_subtrees), following the structure of a binary tree. At this point, a leaf can be either an actual leaf, from the original structure, or a critical node that has no child yet. The second step consists on browsing this list of subtrees to reconstruct a binary tree (rev_subtrees_to_trees). If a tree is complete, it is added to a stack without its annotations (nature of nodes). Otherwise, two subtrees are popped from the stack and then grafted to the current tree whose annotations have been removed: these subtrees are grafted to

the first found critical node (the serialization function, and the first step actually ensure there is only one such node). This grafted tree represents now a complete subtree, and it is stacked at its turn. The deserialization ends when all the subtrees have been grafted and added to the stack. The first, and supposed only one, element of the stack is the binary tree representation of the input linearized tree.

The types serialization and deserializaiton in SyDPaCC are the following:

- serialization: (`BTree`.t A B) → `N` → `list` (Segment A B);
- deserialization : `list` (Segment A B) → `option` (`BTree` A B)
  This function returns
  - → None if the input is not a valid serialized tree
  - → Some t with t a binary tree otherwise

  A list of segments is considered valid if the result is different than None.

To define linearized trees, we use the type LTree that encapsulates the property of well-formation of a list of segments.

```
Definition valid_tree '(tree: list (segment A B)) : bool :=
 is_some(deserialization tree).

Definition LTree A B := { segs:list(segment A B) | valid_tree segs = true }.
```

Distributed trees

Contrary to binary trees, a linearized tree can be easily distributed. A distributed tree is defined as a parallel vector of of list of segments. Each processor has several segments, representing a part of the original tree. Similarly than a LTree, a PLTree must be well-formed. Using the join function of parallel lists (from the module ParList), a parallel tree can be sequentialized into a linear tree. This linear tree must respect the valid_tree property.

```
Definition PLTree A B :=
  { plt: par(list(segment A B)) | valid_tree(ParList.join plt) = true }.
```

Sequential functions

In this section, we will present implementations of *map* and *reduce* on the different representations of binary trees.

On binary trees

On a binary tree, the definitions of *map* and *reduce* are defined recursively, similarly than in BMF. The function map needs two functions as parameters: kL to apply to leaf values, and kN for node values. In other hand, reduce only take one function, k, to reduce a node from a current value, and two sub-reductions. Their Coq definition is given as follows.

```
Fixpoint map (A B C D :Type) (kL:A → C) (kN:B → D) (tree: t A B): t C D :=
 match tree with
 | Leaf n ⇒ Leaf (kL n)
 | Node n l r ⇒ Node (kN n) (map kL kN l) (map kL kN r)
 end.
```

```
Fixpoint reduce (A B:Type) (k: A ∗ B ∗ A → A) (tree: t A B): A :=
 match tree with
 | Leaf n ⇒ n
 | Node n l r ⇒ k (reduce k l, n, reduce k r)
 end.
```

On linearized trees

Map

The map function (Figure 3.2) 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 kL (resp. kN) on the elements marked as leaves (resp. as nodes or critical nodes).

Reduce

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. 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:

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

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

Figure 3.2 – *Formalization of the map function on linearized tree in Coq*

This closure property is written $k =< \phi, \psi_N, \psi_L, \psi_R >_u$.

We represent this property in Coq thanks to a class Closure.

```
Class Closure {A B C}
    (k : A ∗ B ∗ A → A) (phi : B → C) (psiN : A ∗ C ∗ A → A)
    (psiL : C ∗ C ∗ A → C) (psiR: A ∗ C ∗ C → C) :=
 {
  closed : (∀ l b r, k (l,b,r) = psiN (l, phi b, r))
           ∧ (∀ x l y b r, psiN(psiN (x,l,y),b,r) = psiN(x,psiL(l,b,r),y))
           ∧ (∀ x l y b r, psiN(l,b,psiN(x,r,y)) = psiN (x,psiR(l,b,r),y))
 }.
```

The reduction on a linearized tree proceeds in two steps. First, reduce_local applies the functions phi and either psiL or psiR 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. If a segment does not contain a critical node, the local reduction is total. The reduction is partial otherwise. These two cases does not make reduce_local returning the same type of value. That is why, we use the type sum from the standard library.

```
Inductive sum (A B:Type) : Type :=
 | inl : A → sum A B
 | inr : B → sum A B.

Notation "x + y" := (sum x y) : type_scope.
```

If a segment does not contain a critical node, the result will be constructed with inl. In the other case, it will be constructed with inr.

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

```
Definition reduce_local {A B C} (k: (A * B * A) → A)
  (phi : B → C) (psiL: (C * C * A) → C) (psiR: (A * C * C) → C)
  (seg : segment A B) : option (sum A C) :=
match fold_left (opL k phi psiL psiR) (rev' seg) (Some ([],None)) with
| Some (h::t, _) ⇒ Some h
| _ ⇒ None
end.

Definition reduce_global (A C:Type)
  (psiN : A * C * A → A)
  (gt : list(sum A C)) : option A :=
match fold_left (opG psiN) (rev' gt) (Some []) with
| Some (a::_) ⇒ Some a
| _ ⇒ None
end.

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 3.3 – *Formalization of the reduce function on linearized tree in Coq*

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]. This function is described below.

```
Fixpoint map_filter_some A B (f:A→option B) l : list B :=
  match l with
  | [ ] ⇒ [ ]
  | h::t ⇒ match f h with
      | None ⇒ map_filter_some f t
      | Some b ⇒ b::(map_filter_some f t)
      end end.
```

All intermediate results are merged using reduce_global into a single value thanks to psiN. The definitions of each part to reduce a linearized trees are given in Figure 3.3.

```
(* Functions for the Application of reduce *)
Definition k_c: (N * N * N) → N := fun t ⇒ match t with (a,b,c) ⇒ a + b + c end.
Definition phi_c : N → N := fun x ⇒ x.
Definition psiN_c : (N * N * N) → N := fun t ⇒ match t with (a,b,c) ⇒ a + b + c end.
Definition psiL_c : (N * N * N) → N := fun t ⇒ match t with (a,b,c) ⇒ a + b + c end.
Definition psiR_c : (N * N * N) → N := fun t ⇒ match t with (a,b,c) ⇒ a + b + c end.

Instance sum_closure : Closure k_c phi_c psiN_c psiL_c psiR_c.
Proof.
 (* 8 lines *)
Qed.

Definition spec_sumvalues: LTree N N → option N :=
  (reduce k_c) ∘ (map (fun x ⇒ 1) (fun x ⇒ 1)) .
```

Figure 3.4 – *Size of a linearized tree in Coq*

In these definitions, opL and opG corresponds respectively to the functions used in the local and global reductions. opL will either make complete reductions using k, or partial reductions using phi, psiL, and psiR from the closure property. opG will complete the reductions using psiN. They are constructed as follows. In both cases, a stack is used to keep the already reduced values. In the local reduction, an `option N` is also used to describe the type of the last treated element of the segment.

Size

Thanks to map and reduce, we can calculate the size of a tree. First, map is used to replace all the values by 1 within the tree. Secondly, the sum of all values is proceeded using reduce. The code of this example is given in Figure 3.4.

Considering the linear tree of the Figure 3.1, the step-by-step results are the following.

After the application of map, we obtain the same tree with all the values replaced. Since a LTree is a `sig`, it is made from a list of segments, and a proof of validity of this list. Its resulting list of segments is constructed as follows.

```
Definition s1_1 : segment N N:= [(Cr 1)].
Definition s2_1 : segment N N:= [(No 1); (Le 1); (Le 1)].
Definition s3_1 : segment N N:= [(No 1); (Cr 1); (Le 1)].
Definition s4_1 : segment N N:= [(No 1); (Le 1); (Le 1)].
Definition s5_1 : segment N N:= [(No 1); (Le 1); (Le 1)].
```

```
Definition lt_1 : list (segment N N) := [s1_1; s2_1; s3_1; s4_1; s5_1].
```

A local reduction returns an option (N+N) corresponding to the sizes of a subtree. Since we apply reduce_local with map_filter_some, the local reductions will return a list of (N+N).

```
Definition gt : list (N+N) := [inr 1; inl 3; inr 3; inl 3; inl 3].
```

reduce_global is applied to the global structure composed by the values of every local reduction and return Some 13.

Tree skeletons

Since the formalized code as for purpose to be extracted and be run with BSML, we use the **par** type provided in its Coq formalization. A distributed linearized tree is then a parallel vector of a list of segments. We add the well-formed condition to this kind of vector using a boolean predicate valid_tree.

```
Definition PLTree A B :=
  { plt: par(list(segment A B)) | valid_tree(ParList.join plt) = true }.
```

As described above, on a linearized version of a tree, the map function on trees is a map function on lists. Besides the SyDPaCC framework already features a map skeleton on lists. So a parallel map on trees can be obtained by construction using the parallel function provided by SyDPaCC:

```
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).
```

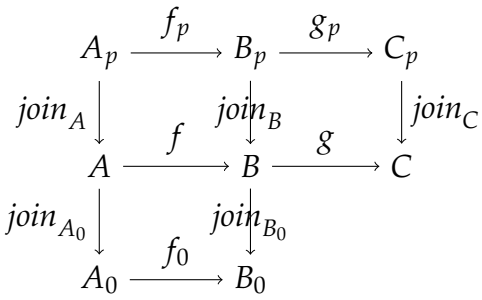
The first part of reduce can be made in parallel using a parallelization of map_filter_some with parfun. The second part cannot be made in parallel. It is only the application of a global reduction, on the list of all values from the previous step. In SyDPaCC, ParList.join transforms a parallel vector of lists into a list (present on all the processors). We perform a global reduction on this list.

```
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).
```

Correspondences

For automatic parallelization of programs, two types of correspondence must be proved: type correspondences and function correspondences. All these details have been presented in Section 2.4.2.

The set of type classes and instances used for automatic parallelization can also be used for a change of sequential representation. Here is a way to compose changes of representations: we call such a composition, vertical composition. The composition of two functions and their correspondences as described above is called horizontal composition. These two kinds of compositions are illustrated in the following diagram:



In this diagram, a type correspondence between A_p and A_0 is established as a composition of the type correspondence A_p to A and then A to A_0. Similarly, a function correspondence between f_0 and f_p is established.

In the context of trees, A_p could be the parallel vector of lists of segments, A the list of segments (linearized tree) and A_0 the type `BTree.t` of trees. Similarly, a function correspondence between f_0 and f_p is established.

Type correspondences

To create a type correspondences between a binary tree and its linear parallel representation, two sub correspondences must be established:

1. a correspondence between a parallel linearized tree (PLTree) and a sequential linearized tree (LTree)

2. a correspondence between a sequential linearized tree (LTree) and a binary tree (`BTree`)

The correspondence 1 is just the same correspondence between a list, and a distributed list.

```
Program Definition join_pltree_ltree ‘(plt: PLTree A B) : LTree A B :=
  ParList.join plt.
```

The second correspondence 1 is more difficult to obtain. We need to find a joining function that from a LTree returns a BTree. In addition, this function must be surjective. A definition of this function is given as follows.

```
Definition join_ltree_btree A B (ltree: LTree A B): BTree.t A B :=
 let Hlt := valid_tree_iff (proj1_sig ltree) (proj2_sig ltree) in
 no_some (exist (fun l⇒l<>None) (deserialization (proj1_sig ltree)) Hlt).
```

In the code above, valid_tree_if is a lemma that states the existence of a result different than None for the deserialization of a LTree.

To prove the surjectivity of join_ltree_btree, we need to show that for all BTree bt, there exists a LTree lt such as join_ltree_btree lt = bt. Since we just have to find one, we picked the most obvious one, serialized with m = 1. In this case, all the nodes will be critical, which is convenient for proving the following lemma, used in the proof of surjectivity.

```
Lemma deserialization_serialization_1:
 ∀ ‘(tree: BTree.t A B),
  deserialization(serialization tree 1) = Some tree.
```

This proof made over here by induction on the tree structure. Since all the nodes are critical, we remove a case: a value cannot have be constructed with No. The other approach would be to use the length of the tree as a value of m. In this case, the case Cr would have been removed. This proof is complete assuming the following lemma. However, this lemma is not proved yet.

```
Lemma rev_subtrees_to_trees_serialization:
 ∀ (m:N) ‘(tree: BTree.t A B),
 ∃ tree’,
  ∀ l stack,
  rev_subtrees_to_trees(segments_to_subtrees (serialization tree m) ++ l) stack =
  rev_subtrees_to_trees l (tree’::stack).
```

Function correspondences

For having automatic parallelization of the functions, different FunCorr instances must be defined. First, a correspondence between a function on a BTree and on a LTree. Then, a correspondence between a function on a LTree and on a PLTree.

Finally, by composition, and from the two previous correspondences, a correspondence between a function on a BTree and on a PLTree.

The correspondences for map are the following:

```
Global Instance map_tree_map_ltree {A B C D} (kL:A→C) (kN:B→D) :
  FunCorr (BTree.map kL kN) (lTree.map kL kN).

Global Instance map_ltree_map_par A B C D (kL:A→C) (kN:B→D):
   FunCorr (LTree.map kL kN) (PLTree.map kL kN).

Global Instance map_tree_map_par '(kL:A→C) '(kN:B→D):
   FunCorr (BTree.map kL kN) (PLTree.map kL kN).
```

These three instances are proved. As discussed in Section 1.2, there are different techniques for proving correctness with formal methods. In one hand, map on parallel trees is obtained by construction, from already proved correct map function on distributed lists. In the other hand, the proof of map_tree_map_ltree is made as posteri. The function on BTree and LTree are defined independently and their equivalence is proved after. Finally, the third instance, map_tree_map_par, is proved by construction from the others.

Similarly, the correspondences for reduce are the following.

```
Global Instance reduce_tree_reduce_ltree (A B C:Type) (k:(A∗B∗A)→A)
       '{H: ClosureU A B C k phi psiN psiL psiR } :
  FunCorr (Some ∘ (BTree.reduce k)) (LTree.reduce k).

Global Instance reduce_ltree_reduce_par '(k:A∗B∗A→A)
      '{Hclose: @ClosureU A B C k phi psiN psiL psiR}:
 FunCorr (LTree.reduce k) (PLTree.reduce k).

Global Instance reduce_tree_reduce_par '(k:A∗B∗A→A)
      '{Hclose: @ClosureU A B C k phi psiN psiL psiR} :
  FunCorr (Some ∘ (BTree.reduce k)) (PLTree.reduce k)(join_B:=(fun x⇒x)∘(fun x⇒x)).
```

As for map_ltree_map_par, the correspondence between LTree.reduce and PLtree.reduce is proved as following: Since only reduce_local is processed in parallel, the global reduction in the two cases are naturally the same. The computation of each segment is independent then distribute them before applying reduce_local, and merge them after is similar than sequentially apply reduce_local to each segment. The instance reduce_tree_reduce_ltree is not proved yet. Assuming reduce_tree_reduce_ltree correct, reduce_tree_reduce_par is proved by composition. Our idea to make this proof is too explicitly defined the global structure a tree, from a linear tree. The reduce function on LTree is decomposed into two parts,

and makes partial calculation thanks to the closure property. The total of partial calculations is the same than the reduction on a binary tree, if the right order is respected. A segment that has a critical node will perform a partial computation during the local reduction, waiting for be branched with the two other subreductions processed in other segments. By explicitly define a path in a linear representation of a tree, we would be able to define which local reduction can be used for a specific moment of the global reduction, and prove that the result is the same than in a binary tree representation.

Discussion

SyDPaCC is, to our knowledge, the only framework that supports the development of verified parallel programs that can be compiled to native code and run on parallel machines. However proof assistants are sometimes used to reason about parallel programs.

Several works are based on a deep embedding of parallel languages or libraries: the syntax and semantics of such languages are modeled using a proof assistant. If it is convenient to have such a formalization to reason about meta properties of the considered language, it is less convenient to write programs than using a shallow embedding as we do for SyDPaCC. For example, Grégoire and Chlipala provide a small parallel language and its semantics and prove correct optimizations of stencil based computations [59]. A subset of Data Parallel C has been formalized using the Isabelle/HOL proof assistant [35]. The tool generates Isabelle/HOL expressions that represent the parallel program rather than actual compilable code. The dependently type language Agda is used by Swierstra to formalize mutable explicitly distributed arrays. He uses this formalization to write and reason about algorithms on distributed arrays: a distributed map, and a distributed sum.

It is, of course, possible to reason about distributed lists, and consider their distribution using the formalization of BSML in Coq. SyDPaCC however allows for the extraction of parallel code, but it does not support mutable data structure. Based on the work of Malecha et al. [95], SyDPaCC may be extended to reason about to extract BSML programs working on mutable data structures.

Note that all the previously cited formalizations consider linear data structures. To our knowledge it is the first time a proof assistant is used to generate parallel programs manipulating trees.

On the implementation side, algorithmic skeletons libraries mostly consider linear data structures such as lists and arrays [7, 24, 37, 83]. One exception is SkeTo[1] [97]: its earlier versions contained tree algorithmic skeletons, but the latest version does not, although recent work considers a new implementation [117].

[1] http://sketo.ipl-lab.org

PySke: A library of skeletons in a mainstream language

4

Contents

In this chapter, we present PySke, a library of skeletons written in Python. After a global presentation of the API, we successively describe how skeletons are constructed for lists and trees. In a final discussion, we present relative works and possible extensions of PySke.

A Python library

The C Message Passing Interface (MPI) standard library has been extended in Python through different libraries, such the *mpi4py* [1] [32, 33, 31] library. Using this extension, we have developed Python skeletons for two data-structures, lists, and trees, with 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. The full library is available at `https://pypi.org/project/pyske/` or can be installed with pip, the package manager of Python, by

```
# pip install pyske
```

Skeletons on Lists

Parallel Lists

The PySke algorithmic skeletons on lists are provided as methods for parallel lists in a class `PList`. The API provides a global view of programs. That is, a PySke program on parallel lists is written as a sequential program on sequential lists. However, the program operates on parallel lists. PySke also offers a class `SList` with additional sequential functions on lists. This is very different from the programming style of MPI and its Python version mpi4py. Both follow the SPMD where the overall program must be thought as a parallel composition of the program being written that depends on the value of a function returning a different value for different processors.

Figure 4.1 illustrates the difficulty to read SPMD program, especially when communications are involved.

[1] `https://github.com/mpi4py/`

```
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)
```

Figure 4.1 – *A mpi4py SPMD Program*

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) depending on the processor identifier.

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 two distinct moment: first the code performed by 0 and then 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.

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 define

Global View

$[0,1,2,3,4,5,6,7,8,9]$

SPMD View

processor	0	1	2	3
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	$[3,3,2,2]$	$[3,3,2,2]$	$[3,3,2,2]$	$[3,3,2,2]$

Figure 4.2 – *Global and SPMD view of* `PList.init(lambda x:x,10)`

on processors 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 4.2 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.

Skeletons

The API 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. The distribution of the output parallel list is the same than `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 `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.

The following program computes the variance of a discrete random variable `X` implemented as 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
```

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 to do so, 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 4.2, 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 distribution 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]`.

Skeletons on Trees

The ill-balanced and irregular structures of Trees make challenging efficient computation. Contrary to lists, the structure of trees is not linear. Instead of proceeding the computation from left-to-right (or right-to-left), it does a from top to bottom (or from bottom to top) computation.

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 on the 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.

Serialization and Distribution

The serialization of trees in PySke is based on the same approach than in SyDPaCC. Here, 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. Tags

are `L` for leaf, `C` for critical node, and `N` for regular node. The example presented in Figure 3.1 can be translated in PySke as follows.

```
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])
```

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 than 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 made in order to have a number of value close to `total_size / nprocs`. If it used without any input, the constructor will return an empty parallel tree.

- A `PTree` can be imported from a text file made with the string output of the type, 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).

SPMD style is also followed for the *implementation* of parallel trees (but the user 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

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]			
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 4.3 – *Global and SPMD view of* `PTree(lt)` (`lt` from Figure 3.1)

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 4.3 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 3.1, on a machine with 4 processors. To simplify the notation, we just represent the `TaggedValue` instances by their values.

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 distribution may be more unbalanced. Matsuzaki et. al. discuss more precisely about this value in [98]. 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 relative decision of splitting.

Skeletons

The tree skeletons described by Skillicorn in [121] represents the base of the tree skeletons implemented in the library. They are provided as sequential functions on all the `Segment` of one or two parallel trees, yielding a new parallel tree.

The pattern on trees follow the ones on lists. The first one, the `map` skeleton, applies two functions to every element of a tree. The need of two functions is due to the non necessary same type for values on nodes and leaves. For a parallel tree `pt`, `pt.map(kL, kN)` is the parallel tress obtained by applying `kL` to each leaf values, and `kN` to each node values, 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))
```

The `reduce` skeleton can be then used on a parallel tree `pt` by `pt.reduce(k, phi, psiN, psiL, psiR)` with `k, phi, psiN, psiL, psiR` respecting the closure property defined in Section 3.2. Its execution necessitates communication and is composed by several steps: Firstly, 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 calculated using `k` and `psiN`. The `reduce` skeleton returns either a single value in the first processor and `None` otherwise.

The Upwards 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 follow:

`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 a 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 with `pid == 0`, and with `psiN`, the actual top values are calculated. The actual top values are redistributed are each `Segment` is finally updated if necessary during one last step using `k`.

The behavior of the Downwards 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)))
```

Another closure property must be defined here. The `dacc` function can be parallelized if there exists phi_L, phi_R, psi_U and psi_D such that:

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

This second 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 with several computation steps. First, each processor computes a local intermediate values with `psiU`, `phiL` and `phiR`. These values correspond to the values to pass to children of critical nodes in a global computation. The intermediate values are gathered to the processor with `pid == 0` and using `psiD` and the initial value of `c`, the first processor computes actual ones. After redistributing them, each processor performs a global downward accumulation can be processes by applying `dacc` locally.

Numbering the nodes with a prefix traversing order of a variable `T` implemented as a parallel tree can be computed as shown in Figure 4.4

Auxiliary functions used for closure properties are obtained using the derivation technique described in [102] The skeletons described above have their sequential implementations for `LTree`. For an instance of distributed tree `pt`, if

```
phi = lambda b: (1, 0, 0, 1)
sum2 = lambda x,y : x + y

def k((ll, ls), b, (rl, rs)): return (ls, ls + 1 + rs)

def psi_n((ll, ls), (b0, b1, b2, b3), (rl, rs)):
  res_1 = b0 * ls + b1 * (ls + rs + 1) + b2
  res_2 = ls + 1 + rs + b3
  return (res_1, res_2)


def psi_l((l0, l1, l2, l3), (b0, b1, b2, b3), (rl, rs)):
  res_2 = (b0 + b1) * l3 + b1 * (1 + rs) + b2
  res_3 = l3 + 1 + rs + b3
  return (0, b0 + b1, res_2, res_3)


def psi_r((ll, ls), (b0, b1, b2, b3), (r0, r1, r2, r3)):
  res_2 = b1 * r3 + b0 * ls + b1 * (1 + ls) + b2
  res_3 = r3 + 1 + ls + b3
  return (0, b1, res_2, res_3)

def gl(c, (bl, bs)): return c + 1

def gr(c, (bl, bs)): return c + bl + 1

def phi_l(b): return 1

def phi_r((bl, bs)): return bl+1

initial = T.map(lambda a : (0,1), lambda x: x)
processed = initial.uacc(k, phi, psi_n, psi_l, psi_r)
prefixed = processed.dacc(gl, gr, 0, phi_l, phi_r, sum2, sum2)
```

Figure 4.4 – *Example of* PySke *skeleton use with prefix numbering application*

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)
```

Rose Trees

The rose trees are also implemented in PySke by the `RNode(v, ts)` constructor where `v` is the value contained in the node and `ts` a list of `RNode` representing children. If `sub == [ ]`, then the current instance is a leaf. All the primitives defined on `BTree` are also defined for `RTree`, plus `lacc` and `racc` representing respectively the Leftward and the Rightward accumulation. Skeletons are not directly defined for this structure. Indeed, Mastsuzaki et. al. have proved that all of these functions can be defined using transformations of `RNode` into `BTree`, (and `BTree` into `RNode` to get an instance of `RNode` as result), `BTree` instance methods [101] and two other ones:

```
  Leaf(a).getchl(c) == Leaf(c)
  Node(b, lb, rb).getchl(c) == Node(l.value, l.getchl(c),
   r.getchl(c))

  Leaf(a).getchr(c) == Leaf(c)
  Node(b, lb, rb).getchr(c) == Node(r.value, l.getchr(c),
   r.getchr(c))
```

For a `rn`, an instance of `RNode`, the transformations can be processed by `bt = rn.r2b()` and `rn = RNode.b2r(bt)`. According to [101], skeletons on `RNode` can be defined from the following equalities:

```
rn.map(f) == RNode.b2r(rn.r2b().map(lambda x:None, f))
rn.map2(rn1, f) == RNode.b2r(rn.r2b().map2(rn1.r2b(), f))
rn.reduce(f, g, unit_f) == rn.r2b().map(lambda x:unit_f, lambda
   x:x).reduce(lambda n,l,r: g(f(n,l),r))
```

The accumulation within the `RNode` instances are more tricky and need more than just quick transformations from `RNode` to `BTree` instances. Following OOP

style, the four accumulators (uacc, dacc, lacc and racc) on RNode can be defined as follows.

```
def uacc(self, f, g, unit_f):
        bt = self.r2b()
        bt2 = bt.map(lambda x: id_f, lambda x: x).uacc(k)
        return RNode.b2r(bt.map2(bt2.getchl(), f))

def dacc(self, f, unit_f):
        bt = self.r2b()
        return RNode.b2r(bt.dacc(f, lambda c,b: c, unit_f))

def lacc(self, f, unit_f):
        bt = self.r2b().map(lambda x:unit_f, lambda x:x)
        return RNode.b2r(bt.uacc(lambda l,n,r: f(n,r)).getchr())

def racc(self, f, unit_f):
        def g(t1, t2):
                (flag1,p1,a1,b1),(flag2,p2,a2,b2) = t1, t2
                if flag1 and flag2:
                        return (True, p1 and p2, f(a1,a2),
   f(b1,a2) if p2 else b2)
                else:
                        return (t1 if flag1 else t2)
        bt = self.r2b()
        gL = lambda c,b:g(c, lambda x : (True, False, None,
   unit_f))
        gR = lambda c,b:g(c, lambda x : (True, True, x, None))
        return RNode.b2r(bt.dacc(gL,gR,(False, True, unit_f,
   None)))
```

We notice the need of specifying the unit element of some operations in the definition above. They must be specified for leaf cases in the resulting BTree instances because they have None as value.

Discussion

Target

PySke is designed for people who wants a trade-off between productivity and efficiency. Python is one of the most used language in the computing world for

its simplicity. Comparing to already libraries, we are not expecting very good performances. *mpi4py*, used in the conception of PySke, is a Python library constructed MPI coded in C. The *mpi4py* library is literally coded in C, then its use needs serialization all the time. These conversions of data from Python types to C supported types, and inversely to exploit the results in Python, lead to a drop in performance. However, a user can easily extend the API with its own functions, and use it on computing mesocenter, i.e. medium clusters. PySke could also be used in academic programs, to introduce parallelism with a skeleton approach.

Other libraries

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 [28] and its successor Sklml offer a set of a few data and task parallel skeletons. Both rely on imperative features of OCaml. parmap [37] a lightweight skeleton library that provides only parallel map and reduce on shared memory machines. BSML [93] is another example of programming library that is function and used to implementation algorithmic skeleton libraries for OCaml [89]. 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-purely extension to the Haskell language [87] that is also used to implement higher-level skeletons [88]. Accelerate is a skeleton library for Haskell that targets GPUs only. The initial proposal of Chakravarty et. al. [20] featured classical data parallel skeletons (map and variants, reduce, scan and permutation skeletons) on multi-dimensional arrays. Besides following an algorithmic skeleton approach, Accelerate optimize the composition of skeletons at *run-time* rather than compile-time, and kernels are also compiled at run-time. For Scala, the Delite [123] 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++ [45, 24], C [7] or Java [34, 19, 86]. 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 superset of a sub-set of the OSL [83] library for C++. Compared to OSL, PySke lacks a skeleton [84] to manage exceptions in parallel. OSL also provides `bh` a parallel skeleton well-suited for bulk synchronous parallelism [81]. PySke does not provide this skeleton yet. OSL is close of the SkeTo library for C++, but SkeTo [40] also provides skeletons for multi-dimensional arrays. The current version of SkeTo does not provide tree skeletons but a previous one did [98]. Recent work considers a new implementation [117] that is not yet included in the current version of SkeTo. Other API, such as SkelGIS [30], are designed for people who does not have knowledge on HPC, but who needs efficient parallel programs for scientific applications such as blood flow arterial network [29].

Other approaches for parallel computation

Parallelization of programs is necessary but not trivial. There exist frameworks to ease their development. MapReduce[36, 80], developed by Google, and written in C++, split the parallel computation into distributed data. This approach is based on the map and reduction functions presented in Section 2.1. With this model, each value is described by a key, and the data with the same keys are treated in the same processor. A MapReduce process is decomposed into three distinct steps:

1. **Map**: each worker applies the same function to its data;
2. **Shuffle**: the results of the previous step are redistributed;
3. **Reduce**: the worker reduce the results, and process output data

Figure 4.5 from [80] gives a view of these computation steps.

Unfortunately, MapReduce is not open-source contrary to its Java implementation Hadoop [5] by Apache. Both have been derived for a more specific computation.

Even if the cores of MapReduce implementations ease parallel computing, their configuration represents a hard task. Contrary to PySke, they need more

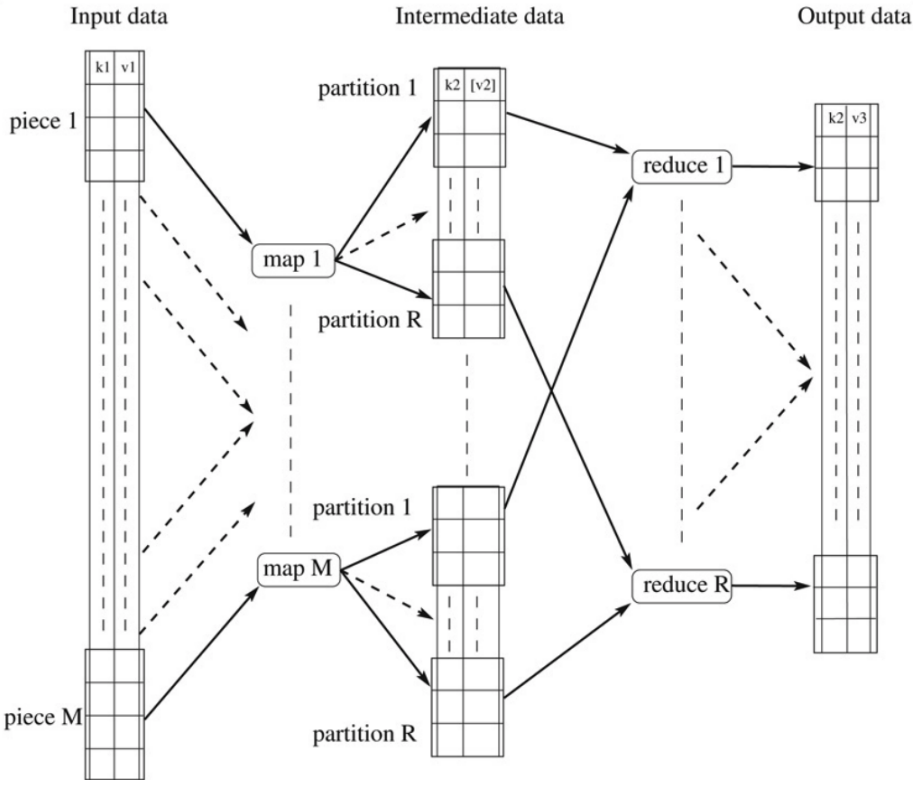


Figure 4.5 – *Computation steps of a MapReduce process*

than the number of processors as parameters. For example, the environment expects type and amount of resources, especially in Cloud computing (e.g., number of virtualized machines, or the number of mapper and reducer). Small changes in the configuration can drastically change the performances. There are studies about finding an automatic configuration of MapReduce environment, such the AROMA project [79] with an approach based on machine learning. PySke has the advantage to be easy to use for every developers. The only needed parameter is the number of cores to use for computation. Besides, our API has the advantage of treating tree structures directly, where MapReduce does not.

Other data structures

For a complete a skeletons API, other structures primitives can be implemented in parallel. First on **arrays**. The structure is already defined in Python and the lists primitives fit for the array calculation. Since arrays are lists with additional property such a fixed size, and a uniformity of element (not necessary with lists in Python), their implementation is more efficient. We are here not only thinking about unidimensional arrays. Algebra defining **matrices** and **multidimensional arrays** as constructive elements [116, 38, 42, 114] are well-suited for parallelism. An global view of these notions are presented in Appendix B.

Graphs are used in many Big Data problems. Networks represent complex data structures, but used in many domains (e.g., social network analysis). The traditional way of graph definition (two sets: vertices, and edges) is not very suited for parallelism. Using a vertex-centric approach instead, as in MapReduce frameworks look better. From MapReduce, Pregel [96, 18, 62], is born for large graph computation. Similarly, Giraph [1, 22, 64, 72] is an Apache project implemented from Hadoop for the same goal. They both are designed using a vertex-centric approach. With this approach, a graph is only defined by a set of vertices, containing information about incoming and/or outcoming edges.

Appendix C describes this approach with more details. We can imagine a class `PGraph`, a list of `CVertex`, representing a parallel subgraph, communicating with the others to operate on the graph content. Primitives described in Section C.2 with additional features such as graph modification (e.g, edges or vertices) or representation transformations (from classical two sets to a vertex-centric graph, and reciprocally) could be implemented.

Parallel programming is often involved in clustering. Make clusters from a large set of data in a reasonable amount of time remains a hard problem to solve. That is why, finding parallel solutions to optimize the computation time

with PySke is an interesting idea. There exist indexing techniques to avoid useless calculations. For example, with a density-based approach for clustering (DBScan [44]), data are stored in grids [61, 135]. A grid is a structure that only gives an index for cells containing object elements. The points from a data set are ordered depending on their relative position. Two closer points will be stored next to each other. This indexing solution is similar than in [57] and [56]. Since in DBScan optimizations, a cell of this grid only need informations about its direct neighbors, we can imagine a distributed data structure `PGrid` containing cells with their objects informations (e.g., coordinates of points), and neighborhood content. The neighborhood content can be all the object informations (e.g., all the coordinates for exact clusters), or only a part (e.g., the number of elements by cell in an approximation method). Appendix D gives more details about different approaches we have already explored to compute clusters using indexing methods.

Examples and experiments 5

Contents

In this chapter, we illustrate the different skeletons we implemented in SyDPaCC and PySke. We first present examples on trees from SyDPaCC, that have been executed using BSML and MPI (Section 5.1). Secondly, we use PySke to define programs on both lists and trees (Section 5.2). We discuss differences between communications of the different used architecture in Section 5.2.2

The examples are executed on a single shared memory machine, called Titan, with two Intel Xeon E5-2683 v4 processors with 16 cores at 2.10 GHz, 256Gb of memory. The used software is the following: Ubuntu Linux 18.04, Python 3.6.7, mpi4py version 3.0.0, OpenMPI version 2.1.1. To test scalability, experiments have also been processed on Monsoon, the HPC cluster of Northern Arizona University. The nodes of the cluster have 16 Intel Xeon cores E5-2620 v2 at 2.10GHz, with a total of 24TB of memory. Individual systems are interconnected via FDR Infiniband at a rate of 56Gbps. The examples are run in the minimum number of nodes, with a maximum of 16 cores by node. For example, from 1 to 16 cores we use 1 node, from 17 to 32 we use 2 nodes, etc.

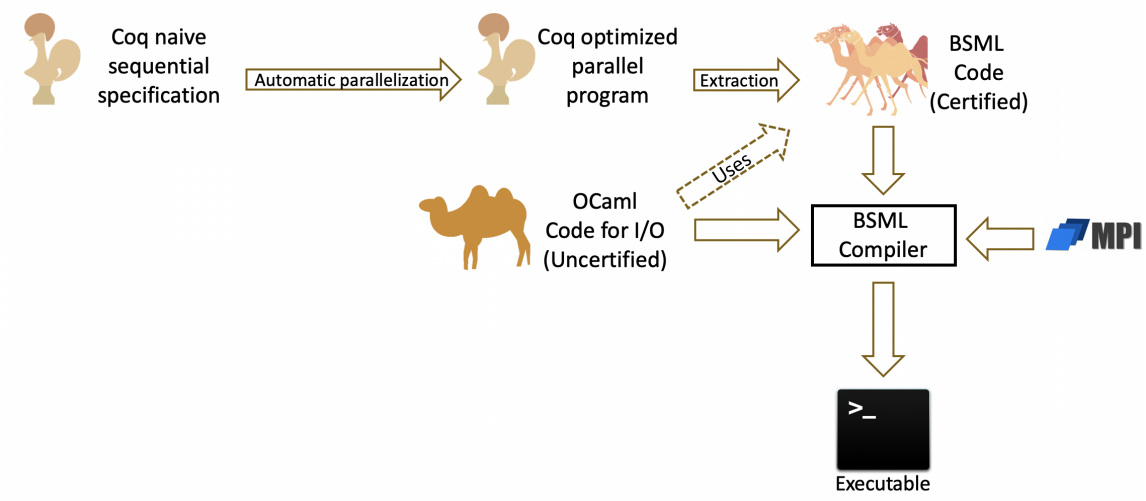


Figure 5.1 – *Process to obtain an executable from a program specification in Coq*

SyDPaCC examples

In this section, we present applications that have been written using SyDPaCC. From a specification written in Coq, and defined correspondences, the function parallel will produce a parallel version of the program. This parallel program and all its dependencies are extracted into OCaml code. To exploit this program, a OCaml script must be written, managing the input and the output of the program. The BSML compiler compiles ML sources files to bytecode executable using MPI. Figure 5.1 illustrates this process.

Height of a Tree

The height of a tree describes the maximum depth that a node can have. The height of a binary tree can be written with a recursive function with a single-bottom up computation:

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

where $x \uparrow y$ computes the maximum of x and y.

This function can be easily defined thanks to the *map* and *reduce* functions:

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

We need to prove that $(\mathsf{fun}\ (x,l,r) \Rightarrow x + (l \uparrow r))$ has the closure property. The following ϕ, ψ_N, ψ_L, and ψ_R are used to show that:

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

The closure property holds only if *map* has been applied before and transformed all the values in the tree into 1. Instead of just using numbers for values in the tree, we specify that the result of the application of map is `BTree N {x : N | 1 = x}`. The functions used in the reduce part must consider this change. Thus the signature of k won't be `(N * N * N) -> N` but `(N * {x : N | 1 = x} * N) -> N`. Same thing for phi whose the signature `N -> (N * N)` is replaced by `{x : N | 1 = x} -> (N * N)`. Without this condition, the closure property does not hold.

To obtain a parallel version of this application, we only need to write `parallel(height).`

Properties Count

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 $BTree\ \alpha\ \beta$ can be implemented as a single-bottom up computation:

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

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

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

and the closure property holds for:

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

The tests have been conducted in the single machine 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 $m = \frac{2\sqrt{N}}{p}$ with N the size of the tree. 30 time measures have been taken on three kinds of tree: balanced, completely unbalanced and with a random shape. Figure 5.2 shows the average computation and 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.

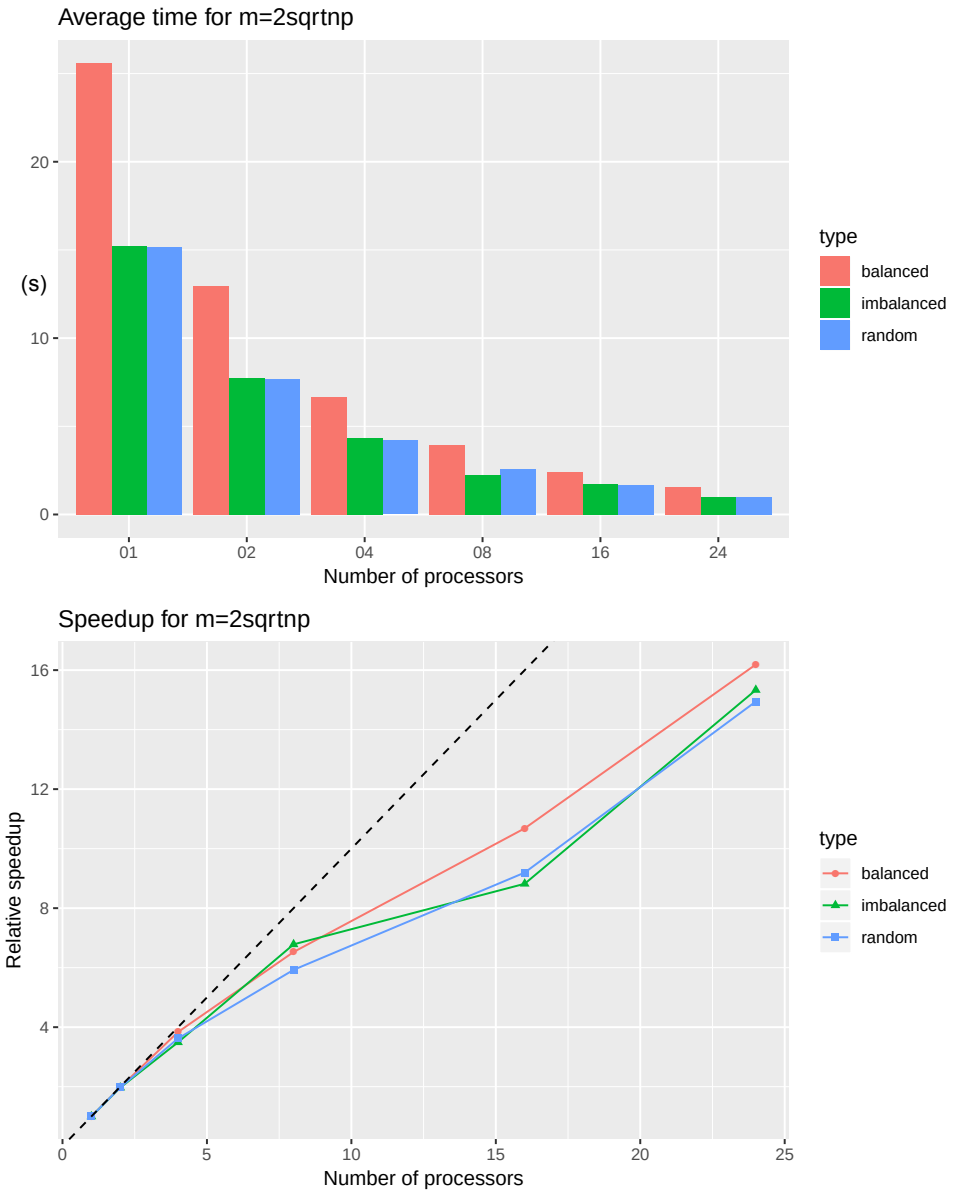


Figure 5.2 – *Performances of correct count using the Titan machine*

PySke examples

We now present applications run with the PySke library on different parallel data structures. They are executed both on the single shared memory machine and on the HPC cluster.

Variance

The first example is the calculation of the *variance* of a data set. The application have been processed using PySke 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. *variance* is defined as follows.

$$\begin{aligned} \textit{variance}\ \text{px} \quad = \quad & (\textit{reduce}\ (+)\ (\textit{map}\ (\lambda\ \text{x} \rightarrow (\text{x - mu})^2)\ \text{px}))\ /\ \text{n} \\ & \textbf{with}\ \text{n} = \textit{size}\ \text{px} \\ & \textbf{and}\ \text{mu} = (\textit{reduce}\ (+)\ \text{px})\ /\ \text{n} \end{aligned}$$

Since the computation of *variance* involves several skeletons and communications, it is a good benchmark to test PySke performances. The PySke code is given page 69.

We processed the variance calculation on a distributed list of $5 * 10^7$ integer elements. Experiments on this application were conducted on the shared memory machine and Figure 5.3 presents the calculation time and the relative speed-up depending on the of processors used for calculation.

The scalability is of course limited depending on the size of the list and the machine resources. We can expect better results with larger datasets. The results show that with more than 32 cores, the relative speed-up decreases. We expect this drop to come from a problem of scheduling: the execution times for a few processes are double than the execution time of most other processes. It is very likely two MPI processes are scheduled on the same core. We did not observe such a phenomenon on the distributed memory platform (Figure 5.4). On the cluster, the results are two times slower. However, there is no scalability problem anymore, even until 256 cores. We discuss about about these performance differences in section 5.2.2

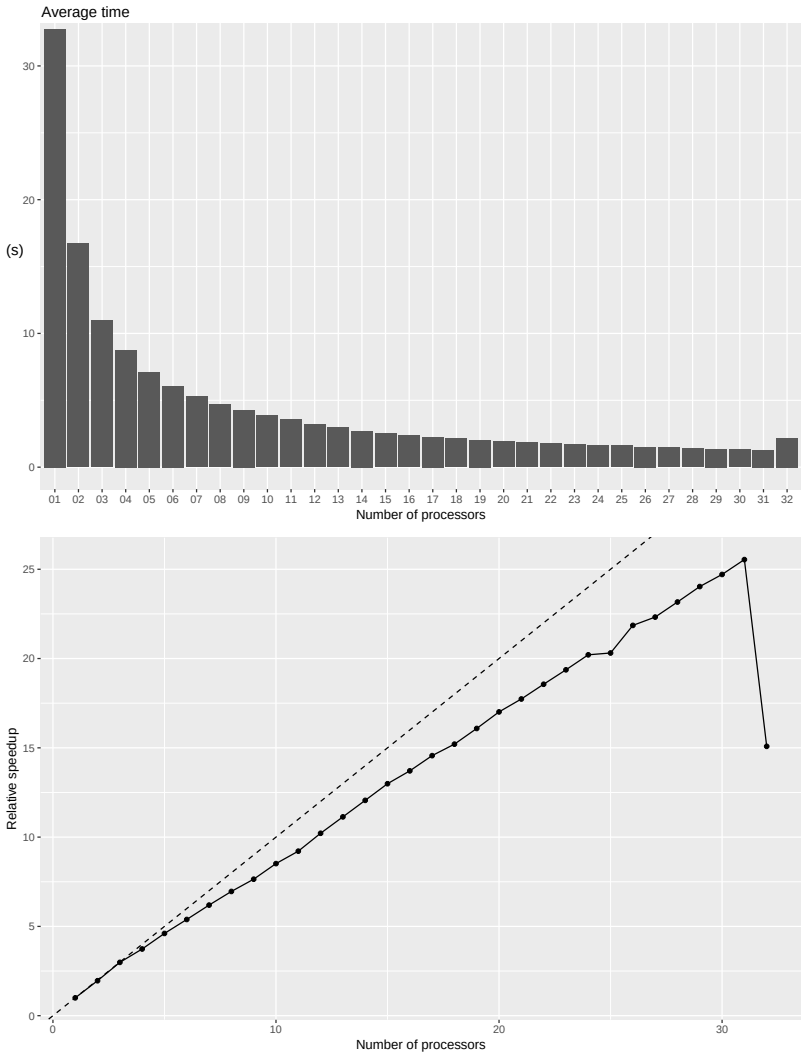


Figure 5.3 – PySke *performances: Variance on Lists using the Titan machine*

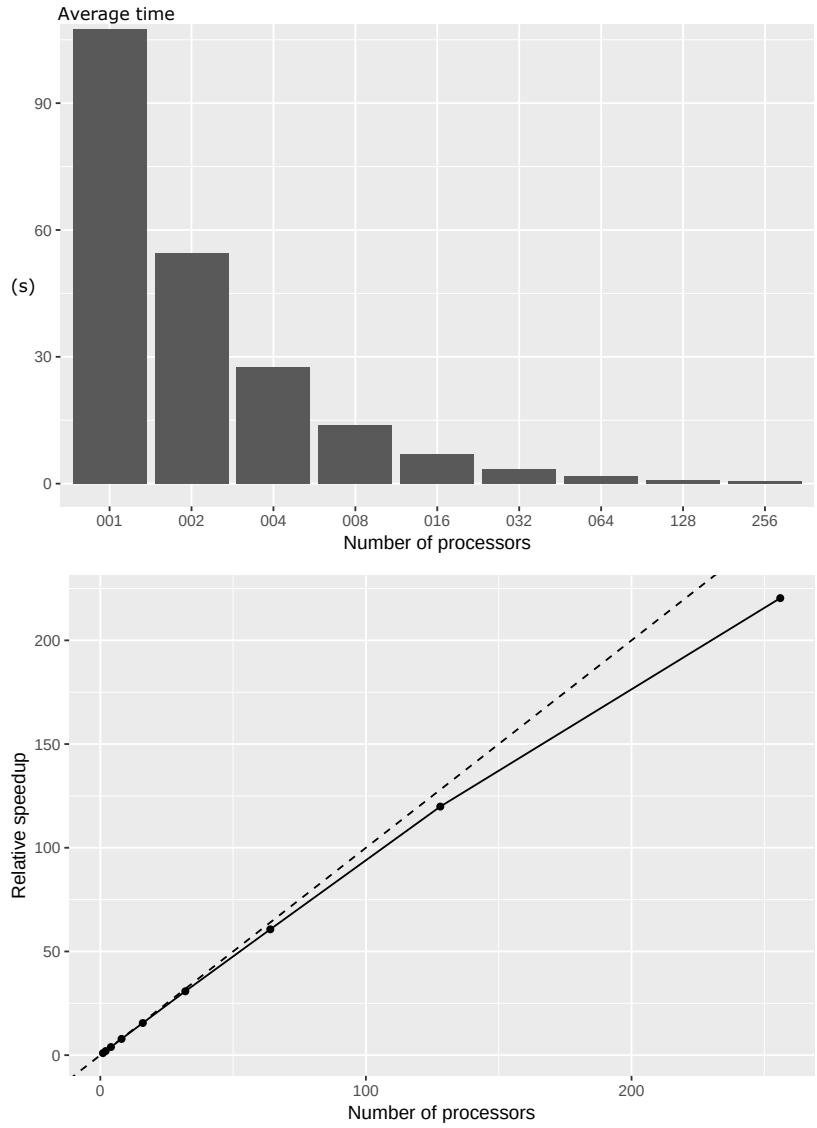


Figure 5.4 – PySke *scalability: Variance on Lists using Monsoon*

Prefix numbering

Numbering elements in a non linear structure such as trees is not trivial. We define *prefix*, a composition of *map*, *uAcc* and *dAcc* for numbering nodes in the prefix traversing order.

$$
\begin{array}{lcl}
\mathit{prefix}\ t & = & \textbf{let}\ t' = \mathit{uAcc}(\mathrm{k}, \mathrm{map}(f, \mathit{id}, t)) \\
 & & \textbf{in}\ \mathit{dAcc}(g_L, g_R, 0, t') \\
 & & \textbf{where}\quad f(a) = (0,1) \\
 & & \qquad\quad k((l_l,l_s),b,(r_l,r_s)) = (l_s,\ l_s + 1 + r_s) \\
 & & \qquad\quad g_L(c,(b_l,b_s)) = c + 1 \\
 & & \qquad\quad g_R(c,(b_l,b_s)) = c + b_l + 1
\end{array}
$$

To be implemented in parallel, closure properties of k and of (g_L, g_R) must be respected. Auxiliary functions for k is given as $k =< \phi, \psi_N, \psi_L, \psi_R >_u$ with

$$
\begin{array}{l}
\phi(b) \quad = (1,0,0,1) \\
\psi_N((l_l,l_s), (b_0,b_1,b_2,b_3), (r_l,r_s)) \\
\qquad = (b_0 \times l_s + b_1 \times (l_s{+}1{+}r_s) + b_2,\ l_s{+}1{+}r_s + b_3) \\
\psi_L((l_0,l_1,l_2,l_3), (b_0,b_1,b_2,b_3), (r_l,r_s)) \\
\qquad = (0,\ b_0 + b_1,\ (b_0 + b_1) \times l_3 + b_1 \times (1{+}r_s) + b_2,\ l_3{+}1{+}r_s + b_3) \\
\psi_R((l_l,l_s), (b_0,b_1,b_2,b_3), (r_0,r_1,r_2,r_3)) \\
\qquad = (0,\ b_1,\ b_1 \times r_3 + b_0 \times l_s + b_1 \times (1{+}l_s) + b_2,\ r_3{+}1{+}l_s + b_3\ )
\end{array}
$$

About g_L and g_R, the closure property is respected by

$$(g_L, g_R) =< \lambda(b_l, b_s) \Rightarrow 1, \lambda(b_l, b_s) \Rightarrow b_l + 1, +, + >_d$$

In PySke, numbering the nodes of a variable `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)
```

We have performed the tests on three 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 linearized using $m = 53600$.

The experiments have been first processed on the Titan machine described before. Figure 5.5 gives a view of these results. Same as the variance application on this machine, the results show that with more than 32 cores, the relative speed-up decreases. We expect the same about scheduling.

To test scalability, they also have been processed on Monsoon, the cluster of Northern Arizona University. Figure 5.6 shows the results of these experiments.

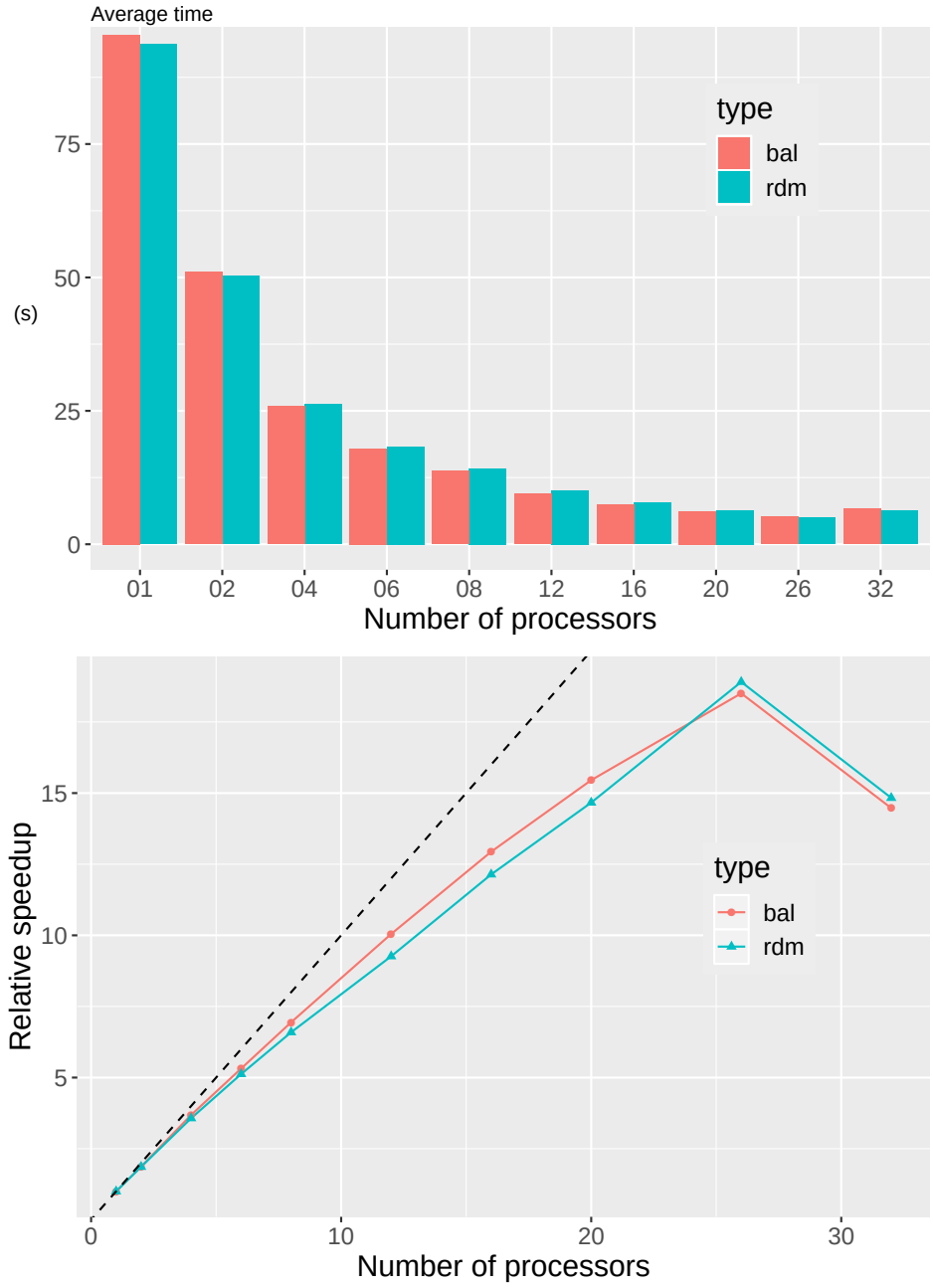


Figure 5.5 – PySke *performances: Prefix numbering on Trees using the Titan machine*

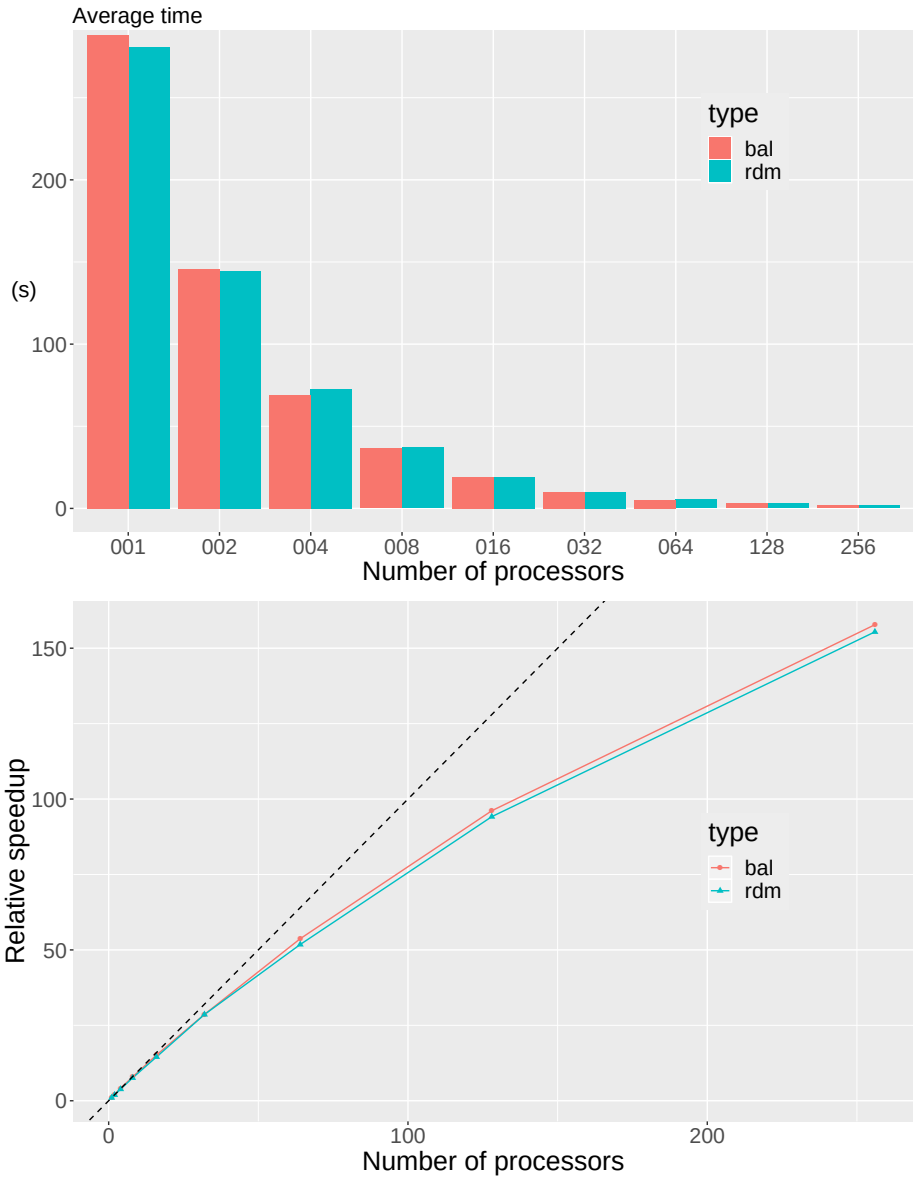


Figure 5.6 – PySke *scalability: Prefix numbering on Trees using Monsoon*

The results of the parallel implementation using PySke skeletons show a good scalability until 128 processors. The computation times of processors are more balanced between for balance tree because the more predictable distribution of the tree. However, even with 256 processing units, the performances are increasing. Python implies an evident performance penalty, especially on Monsoon. For

example, compared to C++ library SkeTo, the same program is slower but the relative speed-up increases similarly. Figure 5.7 shows a comparison of the relative speed-up on same executions of the prefix 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.

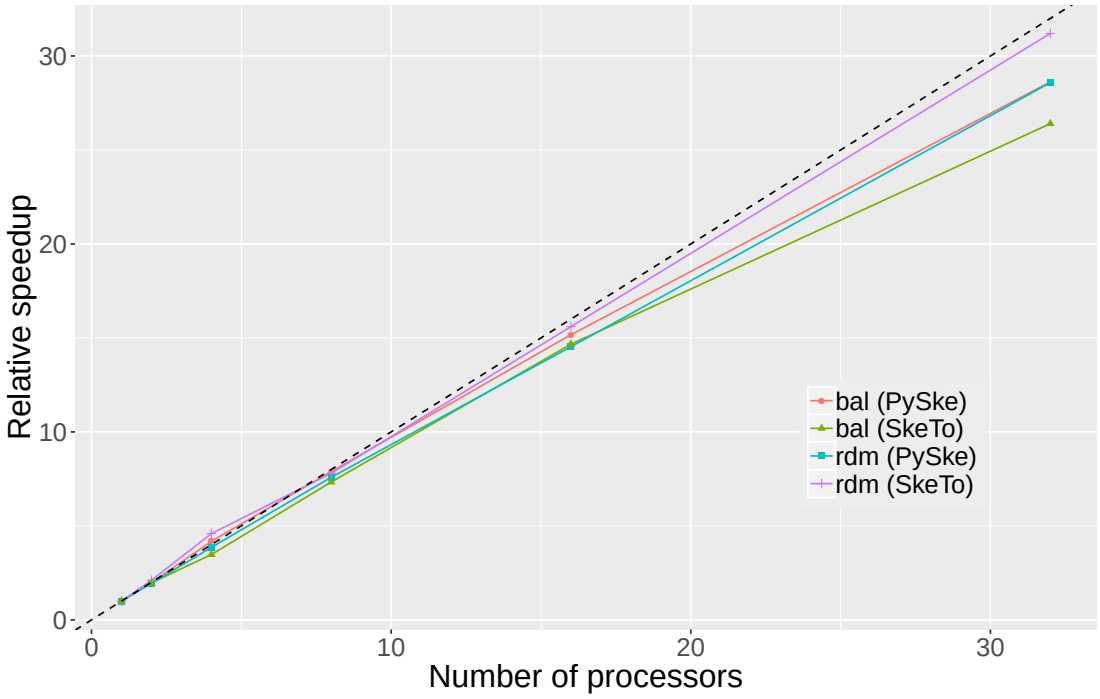


Figure 5.7 – *Comparison of relative Speed-Up of the same program on SkeTo and* PySke

Discussion In both examples, the scalability with the cluster was much better than a single machine. It is interesting to wonder why. The speed of communications between processors are not necessary the same depending on the architecture. The cluster Monsoon interconnects the systems using FDR Infiniband at a rate of 56Gbps. This super fast way of communication tackle the usually loss performances due to communication time.

In a single processor, with several core, the data are communicated almost instantly. However, when a cluster is used for computation, the traditional hardware for communications is not as fast as in a single processor. Even if the machines are individually more effective, the communication time can implies worst global performances. Performances on Monsoon are not as good as they are with

a single machine because processors in the cluster are from an older generation, the nodes of the cluster and the cores of the processors are shared by the different users. Nonetheless, because of the FDR Infiniband, we can expect to never get a scalability decrease while adding processors could have a significant impact on the single machine.

Conclusion and future work 6

Contents

Conclusion

In this thesis, we addressed some difficulties of writing parallel programs without errors. The main objectives were to propose ways to make parallel programming more systematic. We thus proposed two approaches:

- PySke, a library of skeletons in a high-level programming language, providing algorithmic patterns for distributed lists and trees.
- An extension of SyDPaCC allowing to develop parallel programs on parallel linearized trees, correct-by-contruction, and simply by writing sequential programs on binary trees.

We went through examples to illustrate how easy it is to write parallel programs with skeletons. A beginner developer that is not used to parallel programing needs higher-level abstractions. Using skeletons is more attractive for programmers that low-level APIs: at first they can ignore the parallel implementations of the skeletons, and the possible optimization. Each of the proposed approaches has its own advantages.

With SyDPaCC, programmers can write and reason about programs using Coq. In this case, a programmer can write first a sequential implementation of a program, and an implementation using provided primitives. The two versions of the same program can be proved equivalent to ensure that the one written with

skeletons is correct. Alternatively, the programmer can write only the sequential implementation and rely on the provided equivalences to automatically obtain an equivalent correct-by-construction parallel program.

PySke does not provide tools for automatic parallelization, but is very simple to use. Besides, since the API is written in Python, it is easier to write new skeletons. For this reason, PySke provides more parallel patterns, and can be extended faster to handle more parallel problems. Being based on Python, the number of potential users exceeds by far the number of potential users of SyDPaCC.

Future work

Our future works will be focused both on PySke and SyDPaCC.

- The PySke API can be completed with other skeletons. First, the already defined structures can provide more parallel patterns, especially on lists [40, 43, 23]. Also, other structures with their skeletons such as matrices [38], or graphs [41] can be implemented in a future version of PySke. With these additional structures, more applications could be written. For example, the C++ skeleton library SkelGIS [29] performed scientific simulations using skeletons. Providing more structures and more skeletons will increase productivity for sure, but it can also decrease the performances if the skeleton combinations are not optimized. Optimizations can be automatically performed on the specifications using program transformation [53, 108, 114, 70]. Besides, optimizations could be processed using a cost model for the execution of skeletons [2]. On the implementation side, the mpi4py library has been particularly designed and optimized for numeric arrays of the NumPy [112] library. We plan to perform tests using application involving contiguous NumPy arrays. Since we are only using homogeneous parallel structures, arrays would be better to use for computation and communications. The mpi4py library is not the only one that allows parallel programming in Python. We plan to comparatively study the different Python programming libraries[1], both from the runtime performance perspective but also from the productivity perspective. Halstead metrics [63] and similar metrics are well-designed to evaluate the effort needed to write the same program in different ways. It already has been used by Légaux et al. [82], and Coullon et al. [30], and we plan to use such

[1] `https://wiki.python.org/moin/ParallelProcessing`

metrics to provide a comparison of PySke with other parallel programming libraries.

- Secondly, the proposed extension of SyDPaCC is not complete yet. First of all, the correspondences for the *map* and *reduce* skeletons are not all proved yet. Especially *reduce* that needs the formalization of other aspects such as the shape of a global structure from the list of segments. Besides, we plan to design and proof of correctness of two additional skeletons on trees: upwards accumulation and downwards accumulation. We also plan to develop and experiments with several new applications. For the moment our extension is limited to binary trees. It is possible to introduce rose trees and to provide a type correspondence between rose trees and binary trees and associated functions correspondences. Finally, to automatically optimize programs, we plan to implement the diffusion theorem and the third theorems on trees as they already are for lists on SyDPaCC. These theorems are presented in Appendix A

Appendices

Tree theorems

A

Contents

Diffusion theorems

The diffusion theorem can be generalized to binary trees [70, 100]. If h is a function on binary trees defined using $\oplus$ an associative and commutative binary operator, $\otimes$ an associative operator, and four functions k_1, k_2, g_1, g_2, with the following recursive way

$$\begin{aligned} h\ Leaf(a)\ c &= k_1(a,c) \\ h\ Node(b,l,r)\ c &= k_2(b,c) \oplus (h\ l\ (c \otimes\ g_1\ a)) \oplus (h\ r\ (c \otimes\ g_2\ a)) \end{aligned}$$

then it can be transformed into

$$\begin{aligned} h\ x\ c &= (reduce\ (\oplus)\ (map\ k_1\ k_2\ ac)) \\ &\quad \textbf{with} \quad g_L\ c\ b := c \otimes (g_1\ b) \\ &\qquad\quad\ g_R\ c\ b := c \otimes (g_2\ b) \\ &\qquad\quad\ cs := dAcc\ g_L\ g_R\ c\ x \\ &\qquad\quad\ ac := zip\ x\ cs \end{aligned}$$

Third homomorphism theorem

The third homomorphism theorem is also defined on binary trees [108]. However, it necessitates more types and structures than just the basic ones already defined. First, a type $Either\ A\ B$ designing the sum of two sets A and B:

$$Either\ A\ B = Left(A) \mid Right(B)$$

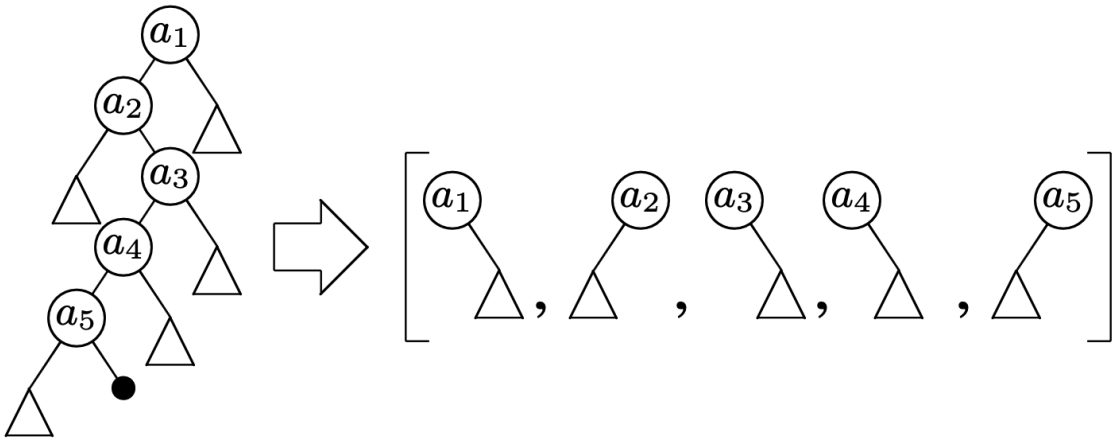


Figure A.1 – *A zipper representing a binary tree using a path from the root to the black leaf*

For shorthand, $Left$ and $Right$ are respectively replaced by **L** and **R**.

The third homomorphism on lists is defined from the possibility of computing a list with leftward and rightward manners. Here, because the trees are represented with a non-linear type, we will talk about upward and downward computations. Using Huet's zippers [71], we cut a binary tree following a path from the root to a selected leaf. We show an example of splitting a binary tree in Figure A.1.

A $Zipper$ of a $BTree\ \alpha\ \beta$ is then a list of $Either\ (\beta, BTree\ \alpha\ \beta)\ (\beta, BTree\ \alpha\ \beta)$ containing the whole tree. $Either$ is here used for its constructors **L** and **R** defining if a subtree is a left or a right children of its ancestor.

Considering $z2t : Zipper \to BTree\ \alpha\ \beta$, a function $h' : Zipper \to B$ is said to be a path-based computation of $h : BTree\ \alpha\ \beta \to A$ if there exists $\psi : B \to A$ such that:

$$\psi \circ h' \quad = \quad h \circ z2t$$

This equation just states an equivalence between calculation on binary trees and zippers.

Because the $Zipper$ representation preserves the hierarchical order of a $BTree$, an upward computation (resp. downward computation) on a binary tree is a leftward computation (resp. rightward computation) on its zipper representation. In other words, a function $h' : Zipper \to B$, the path-based computation of $h : BTree\ \alpha\ \beta \to A$, corresponds to an upward computation if there exists $\oplus : Either\ (\beta, A)\ (\beta, A) \to B \to B$ such that:

$$\begin{array}{lcl} h'\ ([\mathbf{L}(n,t)] +\!\!+ x) & = & \mathbf{L}(n, h\ t) \oplus h'\ x \\ h'\ ([\mathbf{R}(n,t)] +\!\!+ x) & = & \mathbf{R}(n, h\ t) \oplus h'\ x \end{array}$$

A downward computation can be defined similarly using the existence of $\otimes : \mathit{Either}\ (\beta, A)\ (\beta, A) \to B$ such that:

$$\begin{aligned} h'\ (x +\!\!+ [\mathbf{L}(n,t)]) &= h'\ x \otimes \mathbf{L}(n, h\ t) \\ h'\ (x +\!\!+ [\mathbf{R}(n,t)]) &= h'\ x \otimes \mathbf{R}(n, h\ t) \end{aligned}$$

The third homomorphism theorem on trees states that a function h is decomposable into a divide-and-conquer algorithm iff there exists a path-based computation of h that is both downward and upward.

In other words, a function $h' : \mathit{Zipper} \to B$, the path-based computation of $h : \mathit{BTree}\ \alpha\ \beta \to A$ that is both downward and upward can be decomposed using ψ, ϕ, and $\odot$ such that the following equations hold.

$$\begin{aligned} \phi\ \mathbf{L}(n, h\ t) &= h'\ [\mathbf{L}(n,t)] \\ \phi\ \mathbf{R}(n, h\ t) &= h'\ [\mathbf{R}(n,t)] \\ a\ \odot\ b &= h'\ (h'^{\circ}\ a +\!\!+ h'^{\circ}\ b) \end{aligned}$$

Matrix algebra

B

Contents

It is interesting to see how the Bird-Meertens Formalism has been extended for multiple dimensions structures such as matrices. There exist different approaches for representing them.

Two Multidimensional arrays in Abide Trees

Following constructive programming theory [116], a matrix can be defined using three constructors: the singleton $|a|$, representing a matrix with only the element a, and two concatenation operators:

- $u \circ\!\!- d$ representing u above d
- $l \circ\!\!\!\!| r$ representing l above r

From this definition, we can define several representations of the same matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = (|1| \circ\!\!\!\!| |2|) \circ\!\!- (|3| \circ\!\!\!\!| |4|) \text{ or } (|1| \circ\!\!- |3|) \circ\!\!\!\!| (|2| \circ\!\!- |4|)$$

Primitives

The primitives on two-dimensional arrays are intuitive[38, 42]. The first primitive, *map*, applies a function f to every element of the matrix.

$$map\ f \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \dots & x_{mn} \end{bmatrix} = \begin{bmatrix} f\ x_{11} & \dots & f\ x_{1n} \\ \vdots & \ddots & \vdots \\ f\ x_{m1} & \dots & f\ x_{mn} \end{bmatrix}$$

The *reduce* primitive takes two binary operators: $\oplus$ for reducing rows locally, and $\otimes$ to reduce the rows globally.

$$reduce \oplus \otimes \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \dots & x_{mn} \end{bmatrix} = (x_{11} \oplus \dots \oplus x_{1n}) \otimes \dots \otimes (x_{m1} \oplus \dots \oplus x_{mn})$$

The *map2* common operation zip two matrices into a single one using a function f. It can be defined as follows.

$$map2\ f \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \dots & x_{mn} \end{bmatrix} \begin{bmatrix} y_{11} & \dots & y_{1n} \\ \vdots & \ddots & \vdots \\ y_{m1} & \dots & y_{mn} \end{bmatrix} = \begin{bmatrix} f\ (x_{11}, y_{11}) & \dots & f\ (x_{1n}, y_{1n}) \\ \vdots & \ddots & \vdots \\ f\ (x_{m1}, y_{m1}) & \dots & f\ (x_{mn}, y_{mn}) \end{bmatrix}$$

The scan operations *scan* and *rscan* are describing an accumulation of values. The accumulation by *scan* proceeds a left-to-right computation for rows, and a top-to-bottom for columns, to the calculated cell. *rscan* starts its computation from the calculated element, and go through the matrix with a left-to-right computation for rows, and a top-to-bottom for columns to the end of the two-dimensional array. Their definitions are given below.

$$scan \oplus \otimes \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \dots & x_{mn} \end{bmatrix} = \begin{bmatrix} y_{11} & \dots & y_{1n} \\ \vdots & \ddots & \vdots \\ y_{m1} & \dots & y_{mn} \end{bmatrix}$$

$$\textbf{where } y_{ij} = (x_{11} \oplus \dots \oplus x_{1j}) \otimes \dots \otimes (x_{i1} \oplus \dots \oplus x_{ij})$$

$$rscan \oplus \otimes \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \dots & x_{mn} \end{bmatrix} = \begin{bmatrix} z_{11} & \dots & z_{1n} \\ \vdots & \ddots & \vdots \\ z_{m1} & \dots & z_{mn} \end{bmatrix}$$

$$\textbf{where } z_{ij} = (x_{ij} \oplus \ldots \oplus x_{in}) \otimes \ldots \otimes (x_{mj} \oplus \ldots \oplus x_{mn})$$

Homomorphism

Considering two binary associative operators $\oplus$ and $\otimes$, h is a homomorphism on a two-dimensional array if

$$\begin{array}{rcl} h\ |a| & = & f\ a \\ h\ (u \circ\!\!- d) & = & (h\ u) \oplus (h\ d) \\ h\ (l \circ\!\!| r) & = & (h\ l) \otimes (h\ r) \end{array}$$

This homomorphism on matrices is denoted $h = (\!|f, \oplus, \otimes|\!)$.

Generalization

These notions have been generalized for an arbitrary number of dimensions[114]. Instead of considering only two concatenation operations, the operator $+\!\!+_k$ is used for generalizing the concatenation of matrices on the k^{th} dimension. If $d = 1$, then the array is a list, and if $d = 2$, the array is matrix. The equivalence of operations is given below.

Equivalence	#Dimensions	Operators
list	d = 1	$+\!\!+_1 = +\!\!+ = \circ\!\!|$
matrix	d = 2	$+\!\!+_1 = \circ\!\!|$ **and** $+\!\!+_2 = \circ\!\!-$

A function h on d-dimensional multi-dimensional arrays is called a multi-dimensional homomorphism iff there exist d combine operators $\oplus_1, ..., \oplus_d$ such that for each
$k \in [1, d]$,

$$h\ (a\ +\!\!+_k\ b) \quad = \quad h\ a\ \oplus_k\ h\ b$$

Graph with a vertex centric approach

C

Contents

Vertex centric approach

Two main approaches are used for defining graphs. The common approach is to use a pair of sets: V containing the vertices, and E the edges. A graph G is then denoted by $G = (V, E)$. This approach is not very convenient for defining parallel programs. Skillicorn has proposed another approach in [120] to construct graphs.

- $\circ : A \to Graph\ A$ describes a vertex with a single value;
- $\circ\!\!-\!\!\circ : Graph\ A \to Graph\ A \to Graph\ A$ joins two graphs by an edge;
- $\triangle : Graph\ A \to Graph\ A$ adds an additional edge to graph.

This approach is theoretical, and the simple construction does not give enough information about the actual structure of an instance, but it is the foundation of the vertex-centric approach.

Most of the frameworks for parallel computation on graphs [76, 77, 136], such as Google's Pregel [18, 62, 96] and Giraph [1, 22, 64, 72], use the vertex-centric approach, already known as the *think like a vertex* model [115, 113]. With this approach, a graph is only defined by a set of vertices, containing information about incoming and/or outcoming edges.

Fregel

Writing Pregel programs is tedious. Emoto et. al. have proposed a functional model of Pregel, called Fregel [107, 41], to increase the productivity of parallel programs on large-scale graphs. Their functional model is written with Haskell style.

However, Fregel has differences with Pregel. First, the model does not allow the modification of the shape of a graph. It only allows the modification of the content of the vertices. Secondly, in the original model of Pregel, the communications are started by the transmitter. In Fregel it is the contrary. The recipients are peeking the information.

Data types

The functional model proposes three data types to describe graphs.

```
data Vertex a b =
   Vertex { vid :: Int, val :: a, is :: [Edge a b]}
type Edge a b = (b, Vertex a b)
type Graph a b = [Vertex a b]
```

A vertex is described by an integer id, a value and a list of incoming edges. The edges are pairs of weight and source for incoming edges. An example of a graph construction is presented in Figure C.1.

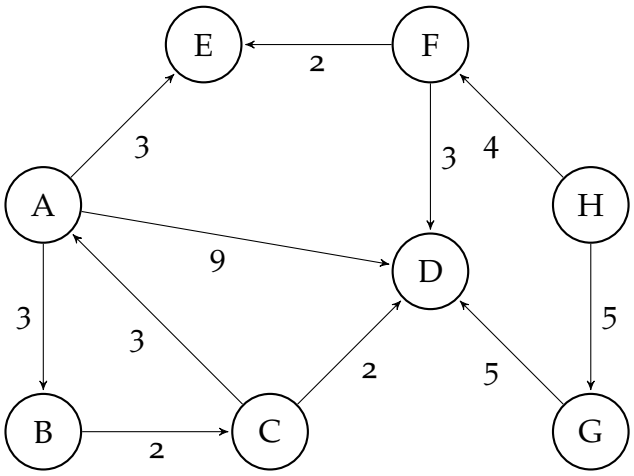
Functional DSL

To modify a graph content, Fregel provides **`makeGraph`** a function that takes two arguments: a graph g and a list of values r, and returns a graph with the same shape of g but with values of r. An example of computation is presented in Figure C.2

Since the status (active/inactive) of a vertex is not represented in Fregel, the termination of a program is modeled by the type `Termination` where `Fix` means a steady state, `Iter` specifies the number of iterations to perform, and `Until` means the program must terminate when the graph respects a given predicate.

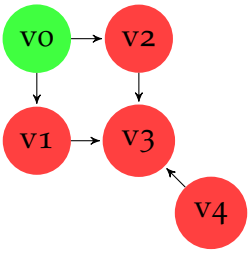
```
data Termination a = Fix | Iter Int | Until (a → Bool)
```

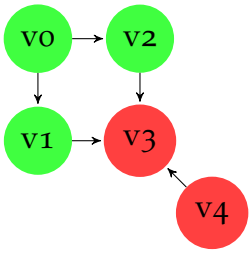
The function `termination` takes the first element of a list that matches with the termination condition. In Haskell, a list is possibly infinite, and can be defined




```
g = let v1 = Vertex 1 A []
      v2 = Vertex 2 B [(3,v1)]
      v3 = Vertex 3 C [(2,v2)]
      v8 = Vertex 8 H []
      v6 = Vertex 6 F [(6,v8)]
      v7 = Vertex 7 G [(5,v8)]
      v5 = Vertex 5 E [(2,v6),(3,v1)]
      v4 = Vertex 4 D [(2,v3),(3,v5),(5,v7),(9,v1)]
  in [v0, v1, v2, v3, v4, v5, v6, v7, v8]
```

Figure C.1 – *Example of graph construction in Fregel*





g := [v0, v1, v2, v3, v4] *g'* := *makeGraph g l* = [v0, v1, v2, v3, v4]
l := [true, true, true, false, false]

Figure C.2 – *Example of makeGraph computation*

as a stream. Even using `termination`, a function on graphs not necessarily terminates.

```
termination :: Eq a ⇒ Termination a → [a] → a
termination Fix xs =
  fst . head . dropWhile (\(a,b) → (a /= b)) $
  zip xs (tail xs)
termination (Iter n) xs = head (drop n xs)
termination (Until p) xs =
  head $ dropWhile (not . p) xs
```

The four functions of Fregel are the following:

- **`gmap`** and **`gzip`**: respectively equivalents of *map* and *zip* defined on the other data structure;
- **`giter`**: An iterative function that applies **`gmap`** until termination;
- **`fregel`**: An iterative function more efficient than **`giter`** that does not manipulate the graph but only its values. From an iteration to another, **`fregel`** keeps previous and current values, and an input high-order function `step` to compute the values for the next step. When the termination condition is reached, the input graph is modified and finally returned.

These four functions are described as follows.

```
gmap :: (Vertex a b → r) → Graph a b → Graph r b
gmap f g = makeGraph g (map f g)

gzip :: Graph a1 b → Graph a2 b → Graph (Pair a1 a2) b
gzip g1 g2 =
   makeGraph g1 (map2 (\u v → Pair (val u) (val v)) g1 g2)
```

```
giter :: (Eq r, Eq b) ⇒ (Vertex a b → r) → (Graph r b → Graph r b) →
   Termination (Graph r b) → Graph a b → Graph r b
giter init iter term g =
   let g0 = gmap init g
      gs = iterate iter g0
   in termination term gs

fregel :: (Vertex a b → r) → (Vertex a b → (Vertex a b → r) →
   (Vertex a b → r) → r) → Termination (Graph r b) → Graph a b →
   Graph r b
fregel init step term g =
   let rs0 = map init g
      f rs_old = let    rs_new = map (\v → step v prev curr) g
                  prev u = rs_old !! (vid u)
                  curr u = rs_new !! (vid u)
                  in    rs_new
      rss = iterate f rs0
   in termination term (map (makeGraph g) rss)
```

Density-based spatial clustering of applications with noise (DBScan)

D

Contents

DBScan

Many domains use Density-Based Spatial Clustering of Applications with Noise (DBScan) [44]. The performance of the DBScan algorithm has been improved thanks to different techniques such as parallel implementation based on indexing techniques [61, 135].

The points are clustered using two parameters: (i) ϵ, an arbitrary distance used to find the neighborhood of a point, and (ii) $minPts$, the minimum number of points within ϵ-distance to create a cluster. The most common approach is to compare the distance points to points to define which ones can be in the same cluster. The main idea is, from a point in a database, look for its ϵ-neighborhood, and check that there are at least $minPts$ points. If there is, the point is a member of a cluster. To find every point of this cluster, we record the ϵ-neighborhood point-by-point from the ones which are already in the same cluster. If a point doesn't have $minPts$ ϵ-neighbors, it is considered as noise. The distance calculation can be made in different ways, but it used to be the Euclidian natural distance:

$$dist_{euclid}(A, B) = \sqrt{\sum_{i=1}^{dim}(x_{Ai} - x_{Bi})^2}$$

where x_{Ai} (resp. x_{Bi}) is the value of the point A (resp. B) for the i^{th} dimension. An outline of the expression of DBScan [44] is presented in Algorithm 1. The algorithm takes several variables as input: (i) the dataset D of points to be clustered; (ii) the distance ϵ to find neighborhood of a point; (iii) the minimum number of points, $minPts$, to consider a cluster; and (iv) a function to calculate distance between points.

To ensure better performance for the research neighbors, an index I of the points used to be created before the execution of DBScan. It is then passed as an argument of the function and is used in *NeighborSearch*. The algorithm examines all points P in D that have not been visited yet ($label(P) \neq undefined$). The neighbors of P are stored in a set N. If N is large enough ($|N| \geq minPts$), P is considering as starting a new cluster. P is a noisy element otherwise. All the neighbors are explored. Two cases can happen. A neighbor Q is either already in a cluster, and then we don't treat it, or Q is now a member of the same cluster than P. If Q is not noise, and has enough neighbors to form a cluster, its neighbors are also members of the same cluster. The output of the algorithm is a label for each point of the dataset. It is either *Noise* or the identifier of a cluster.

Indexing

Since the datasets are extensive, it is convenient to make the data more accessible. We outline in this Section an indexing method to facilitate the data access and remove useless calculation. The indexing is similar than in [57] and [56] solutions. A grid index is appropriate to distance calculation. By making cells with ϵ side squares, we are sure that all the points within ϵ-distance are contained in the direct neighborhood of the current cell. This approach is similar to the method used for trajectory calculations [58]. On a two-dimensional dataset, the points from the points dataset D are defined by two coordinates: x and y. The two-dimension grid is represented with several arrays. An example is given in Figure D.1.

The array A is a lookup array of the dataset D, respecting the spatial position of the elements. In other words, the points are sorted to make them near other spatially close elements. A linearized id defines each cell. The array B contains the id of the non-empty cells of the grid. Array G, which has the same length of B is a lookup array giving information about the contained points of the cells. For each cell, contained at the position h in B, two values are defined: A^h_{min} and A^h_{max}. These values define the position in A of the points contained in the cell h.

Algorithm 1 The DBSCAN Algorithm

```
1:  procedure DBSCAN(D, ϵ, minPts, dist)
2:      C ← 0
3:      for each points P ∈ D do
4:          if label(P) ≠ undefined then
5:              continue
6:          end if
7:          N ← NeighborSearch(D,ϵ,P,dist)
8:          if |N| < minPts then
9:              label(P) ← Noise
10:             continue
11:         end if
12:         C ← C + 1
13:         label(P) ← C
14:         S ← N\{P}
15:         for each points Q ∈ S do
16:             if label(Q) = Noise then
17:                 label(Q) ← C
18:             end if
19:             if label(Q) ≠ undefined then
20:                 continue
21:             end if
22:             label(Q) ← C
23:             N ← NeighborSearch(D,ϵ,Q,dist)
24:             if |N| ≥ minPts then
25:                 S ← S ∪ N
26:             end if
27:         end for
28:     end for
29: end procedure
```

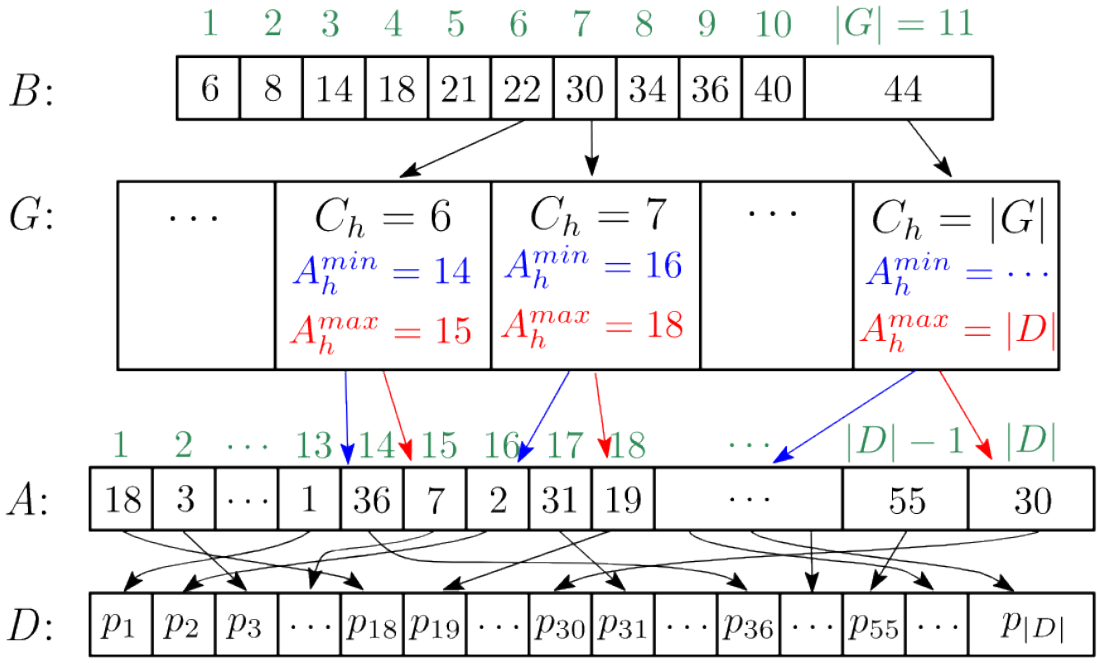


Figure D.1 – *Indexing of points from a dataset D. A is the lookup array to D, G the index array and B the lookup array of G [**?**]*

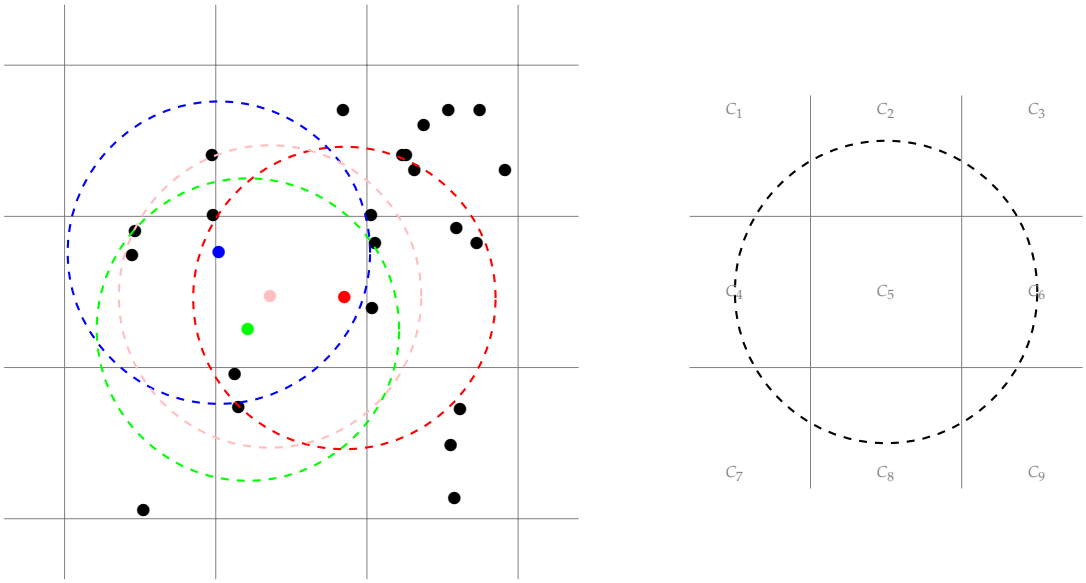


Figure D.2 – *Example of overlapping areas from points of cell with the cells of the neighborhood*

For example, considering the example in Figure D.1, the cell with the linearized id 22, at the 6^{th} position, contains the points from 14 to 15 in A, that is the points p_{26} and p_7.

DBScan by approximation

We propose another approach based on approximation that can be easily parallelized. The decision of merging the cluster of a given cell, and the one of a neighbor cell which contains n_c point objects are made with the following formula:

$$d = \frac{A_o}{A_c} * n_c * p$$

with A_o, the overlapping area of a circle of radius ϵ from a point and the cell, and A_c, the area of a cell, defined by ϵ^2. Because the distribution of points is not necessarily uniform, we need a probabilist constant p to make our decision more or less strict. If there exists a point within the current analyzed cell such that $d \geq minPts$, the neighbor cell is considered in the same cluster. Figure D.2 shows an example of the possible overlapping areas from points within a cell.

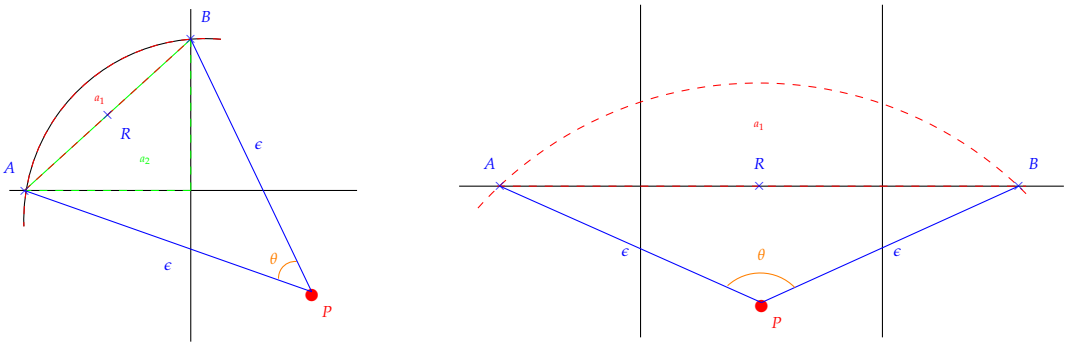


Figure D.3 – *Two cases of overlapping area calculation*

The calculation of A_o, the overlapping area, depends on the cell position. We consider two types of cell: the corners, and the centered cells. In Figure D.2, C_1, C_3, C_7, C_9 are corners, while C_2, C_4, C_6, C_8 are centered cells. Figure D.3 presents two simple cases when these overlapping areas are defined. They are not necessarily defined, depending on the reference point. If one of the coordinates corresponds to a limit of the cell, the opposite neighbor wont be covered by its circle. In both cases, considering A=(x_A, y_A), B=(x_B, y_B) and P=(x_P, y_P), R is defined by
$R = (x_R, y_R) = (\frac{(x_A+x_B)}{2}, \frac{(y_A+y_B)}{2})$. We consider h the distance from P to R. Then, the area a_1 is equal to

$$a_1 = \int_{h\ sin(\theta/2)}^{-h\ sin(\theta/2)} \int_{\sqrt{h^2-x^2}}^{h\ cos(\theta/2)} dy\ dx$$
$$\text{with } \theta = 2 * arccos(\frac{h^2}{\epsilon})$$

which can be simplified to

$$a_1 = \frac{1}{2}h^2(\theta - sin(\theta))$$

The area a_2 is calculate using the triangle area formula. Finally, the overlapping area of a corner is simply defined by $a_1 + a_2$.

Calculate the overlapping area of a centered cell is equivalent to calculate the area of global side and remove the two corner areas.

For example, using Figure D.2, calling A_i the overlapping area within C_i, we have the following result

$$A_2 = overlapping((C_1 \cup C_2 \cup C_3)) - A_1 - A_3.$$

To design an approximation for DBSCAN, we have used the indexing method combined with the overlapping area. The specific entry parameters of the algorithm are the grid resulting from the indexing method, and a probabilistic factor for the decision taking.

We start by making a set for each cell, to contain the linear ids of the cells which will be in the same cluster. To make clusters, every non-empty cell is considered globally. The goal is to merge cells that have enough points (greater than *minPts*), within an ϵ distance from a point, in the same cluster. If two cells appear to be in the same group, they have to merge their cluster. That is, all the cells already defined as being in the same cluster than the current analyzed one, must update their set of ids by adding the new one. At the end of the execution, each cell has a set of several ids. To identify the different clusters, the maximum linear id of each set is kept. This choice is totally arbitrary. It could be the minimum one.

An outline of the algorithm is presented in Algorithm 2. The variables taken as input are: (i) the index G as a grid; (ii) a probabilistic factor for the decisions; (iii) the distance ϵ to find neighborhood of a point; (iv) the minimum number of points *minPts*. We can notice that we don't need to calculate distances anymore, then the *dist* parameters from Algorithm 1 has been removed.

Other approaches

Many approaches have addressed improvement of DBScan performances [12, 55, 57, 65, 134]. These optimizations are made using parallelism. In [65], a MapReduce [36] implementation of DBScan is proposed. The program is split into two parts. First, small local clusters are made using split data distributed on the nodes. A reduction of these results are made in a second part to get bigger clusters. The clusters are first merged and then relabeled to obtain a final result. It is common to make several DBScan execution in science fields (e.g., space physics and aeronomy). The execution variants differ by their input parameters. It appears that several results can be reused during the variant computations. [55] presents very good optimizations for DBScan based on the use of commonalities on multithreading programs. Most of the optimizations are based on GPU computation to take advantage of the GPU architecture. However, [57] presents a grid-based hybrid approach of DBScan, using both GPUs in conjunction with multicore CPU. Two GPU kernels are used. The first one is used to compute the ϵ-neighborhood of the points without using shared memory. Hybrid-DBSCAN

Algorithm 2 Approximation of DBSCAN Algorithm

```
1: procedure APPROXIMATE_DBSCAN(G, factor, ϵ, minPts)
2:     for each non-empty cell C ∈ G do
3:         cluster(C) ← {C}
4:     end for
5:     for each non-empty cell C ∈ G do
6:         N ← neighbors(C)
7:         for each point P ∈ C do
8:             A ← overlapping_around(P)
9:             for i in 0 .. |A|-1 do
10:                np ← N[i].nb_points * A[i]/(ϵ^2) * factor
11:                if np ≥ minPts then
12:                    cluster(C) ← cluster(C)
                                    ∪ {N[i]};
13:                    for each cell C_n ∈ cluster(C) do
14:                        cluster(C_n) ← cluster(C_n)
                                          ∪ {N[i]};
15:                    end for
16:                end if
17:            end for
18:        end for
19:    end for
20:    for each non-empty cell C ∈ G do
21:        cluster(C) ← max_id(cluster(C))
22:    end for
23: end procedure
```

takes advantage of the shared memory on the GPU to page the cells, before making distance calculations. With CUDA-DClust [12], Böhm et al. take advantage of the extremely high parallelism of the GPU, and its low cost of memory transfer. The presented algorithm starts by creating chains of points in parallel on the GPU. The algorithm keeps track of collisions, that is two chains belong to the same cluster. The chains are finally merged into clusters based on the collisions. This approach is very similar than Yaobin et al.'s in [65] This main idea is reused in the Mr.Scan implementation [134], an algorithm which performs kernel optimizations by reducing host-GPU interaction. Another optimization for DBScan implementation based on GPU calculation is discussed in [4]. An index of the data is generally used on these different approaches to reduce the time of computation, and help to remove useless calculation. On a grid-based algorithm approach, the points are stored into cells. The points from a cell are compared to the ones in the neighbor cells. The simple approach is to compare each cell with all of its neighbors. [56] presents an optimization of these comparisons, by non-duplicating the computation, based on the symmetry of the calculation. Gan and Tao proposed an approximate sequential solution [50, 51], faster to compute than DBScan, and with outstanding accuracy.

Bibliography

[1] Apache Giraph. http://giraph.apache.org. 81, 111

[2] Marco Aldinucci, Marco Danelutto, Jan Dünnweber, and Sergei Gorlatch. Optimization techniques for skeletons on grids. In Lucio Grandinetti, editor, *Grid Computing The New Frontier of High Performance Computing*, volume 14 of *Advances in Parallel Computing*, pages 255 – 273. North-Holland, 2005. 98

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, Summer 2007. 2

[4] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. G-DBSCAN: A GPU accelerated algorithm for density-based clustering. *Procedia Computer Science*, 18:369 – 378, 2013. 2013 International Conference on Computational Science. ix

[5] Apache Software Foundation. Apache Hadoop. https://hadoop.apache.org/. 79

[6] Roland Backhouse. An exploration of the bird-meertens formalism. Technical report, In STOP Summer School on Constructive Algorithmics, Abeland, 1989. 8, 17

[7] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In J. C. Cunha and P. D. Medeiros, editors, *11th International Euro-Par Conference*, LNCS 3648, pages 761–770. Springer, 2005. 64, 79

[8] Richard Bird and Oege de Moor. The algebra of programming. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*, pages 167–203, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc. 8

[9] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst.*, 6(4):487–504, October 1984. 8

[10] Richard S. Bird. An introduction to the theory of lists. In Manfred Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. 21

[11] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press, Inc., New York, NY, USA, 2004. 27

[12] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. Density-based clustering using graphics processors. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 661–670, New York, NY, USA, 2009. ACM. 123, ix

[13] Wadoud Bousdira, Frédéric Gava, Louis Gesbert, Frédéric Loulergue, and Guillaume Petiot. Functional parallel programming with revised Bulk Synchronous Parallel ML. In *2010 First International Conference on Networking and Computing*, pages 191–196, Nov 2010. 28

[14] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. 30

[15] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013. 30

[16] J. N. Buxton and B. Randell, editors. *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970. 7

[17] Dominique Cansell and Dominique Mery. Foundations of the B method. *Computers and Informatics*, 22:31 p, 01 2003. 8

[18] Ludovic A. R. Capelli, Zhenjiang Hu, and Timothy A. K. Zakian. ipregel: A combiner-based in-memory shared memory vertex-centric framework. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, ICPP '18, pages 33:1–33:10, New York, NY, USA, 2018. ACM. 81, 111

[19] Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par Parallel Processing*, volume 4641 of *LNCS 4641*, pages 72–81. Springer, 2007. 79

[20] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod 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*, pages 3–14, 2011. 78

[21] Blundell Charles, Yee Whye Teh, and Katherine A. Heller. Bayesian rose trees. *CoRR*, abs/1203.3468, 2012. 25

[22] Avery Ching. Scaling Apache Giraph to a trillion edges. August 2013. 81, 111

[23] P. Ciechanowicz, M. Poldner, and H. Kuchen. The Münster Skeleton Library Muesli – A Comprenhensive Overview. Technical Report Working Paper No. 7, European Research Center for Information Systems, University of Münster, Germany, 2009. 98

[24] Philipp Ciechanowicz and Herbert Kuchen. Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 108–113, 2010. 64, 79

[25] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989. 8, 9

[26] Murray Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. Technical report, Proceedings of Parco 93. Elsevier Series in Advances in Parallel Computing, 1993. 21

[27] Murray Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5:191–203, 1995. 12, 21

[28] R. Di Cosmo, Z. Li, S. Pelagatti, and P. Weis. Skeletal Parallel Programming with OcamlP3l 2.0. *Parallel Processing Letters*, 18(1):149–164, 2008. 78

[29] Helene Coullon, Jose-Maria Fullana, Pierre-Yves Lagree, Sebastien Limet, and Xiaofei Wang. Blood flow arterial network simulation with the implicit

parallelism library skelgis. *Procedia Computer Science*, 29:102 – 112, 2014. 2014 International Conference on Computational Science. 79, 98

[30] Hélène Coullon and Sébastien Limet. Algorithmic skeleton library for scientific simulations: Skelgis. In *2013 International Conference on High Performance Computing Simulation (HPCS)*, pages 429–436, July 2013. 79, 98

[31] Lisandro Dalcin, Rodrigo Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using Python. *Advances in Water Resources*, 34(9):1124 – 1139, 2011. New Computational Methods and Software Tools. 66

[32] Lisandro Dalcin, Rodrigo Paz, and Storti Mario. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108 – 1115, 2005. 66

[33] Lisandro Dalcin, Rodrigo Paz, Storti Mario, and Jorge D'Elia. MPI for Python: Performance improvements and mpi-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655 – 662, 2008. 66

[34] M. Danelutto and P. Dazzi. Joint Structured/Unstructured Parallelism Exploitation in Muskel. In *International Conference on Computational Science (ICCS)*, LNCS. Springer, 2006. 79

[35] M. Daum. Reasoning on Data-Parallel Programs in Isabelle/Hol. In *C/C++ Verification Workshop*, 2007. 63

[36] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. 79, 123

[37] Roberto Di Cosmo and Marco Danelutto. A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml. In *International Conference on Computational Science (ICCS)*, volume 9, pages 1837–1846. Elsevier, 2012. 64, 78

[38] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi. A compositional framework for developing parallel programs on two-dimensional arrays. *International Journal of Parallel Programming*, 35(6):615–658, Dec 2007. 81, 98, 108

[39] Kento Emoto, Frédéric Loulergue, and Julien Tesson. A Verified Generate-Test-Aggregate Coq Library for Parallel Programs Extraction. In *Interactive*

Theorem Proving (ITP), number 8558 in LNCS, pages 258–274, Wien, Austria, 2014. Springer. 42

[40] Kento Emoto and Kiminori Matsuzaki. An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo. *Int J Parallel Prog*, 2013. 79, 98

[41] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, Akimasa Morihata, and Hideya Iwasaki. Think like a vertex, behave like a function. a functional dsl for vertex-centric big graph processing. In *ICFP*, 2016. 98, 112

[42] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Surrounding theorem: Developing parallel programs for matrix-convolutions. In Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner, editors, *Euro-Par 2006 Parallel Processing*, pages 605–614, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 81, 108

[43] 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. 98

[44] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996. 82, 117, 118

[45] J. Falcou, J. Sérot, T. Chateau, and J.-T. Lapresté. Quaff: Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32:604–615, 2006. 79

[46] Donald Firesmith. Using V models for testing, Nov 2013. vii, 6

[47] Robert W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993. 7

[48] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sep. 1972. 3

[49] Message Passing Interface Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994. 27

[50] Junhao Gan and Tao Yufei. DBSCAN revisited: Mis-claim, un-fixability, and approximation. In *SIGMOD Conference*, pages 519–530. ACM, 2015. ix

[51] Junhao Gan and Tao Yufei. On the hardness and approximation of euclidean DBSCAN. *ACM Trans. Database Syst.*, 42(3):14:1–14:45, 2017. ix

[52] Jeremy Gibbons. An introduction to the bird-meertens formalism. Presented at New Zealand Formal Program Development Colloquium Seminar, Hamilton, November 1994. 8, 17

[53] Jeremy Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657—-665, May 1995. 20, 21, 98

[54] Jeremy Gibbons. Calculating functional programs. In Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 148–203. Springer-Verlag, 2002. 8

[55] Michael Gowanlock, David M. Blair, and Victor Pankratius. Exploiting variant-based parallelism for data mining of space weather phenomena. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 760–769, May 2016. 123

[56] Michael Gowanlock and Ben Karsin. GPU accelerated self-join for the distance similarity metric. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2018, Vancouver, BC, Canada, May 21-25, 2018*, pages 477–486. IEEE Computer Society, 2018. 82, 118, ix

[57] Michael Gowanlock, Cody M. Rude, David M. Blair, Justin D. Li, and Victor Pankratius. Clustering throughput optimization on the GPU. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 832–841, May 2017. 82, 118, 123

[58] Michael G. Gowanlock and Henri Casanova. Distance threshold similarity searches: Efficient trajectory indexing on the GPU. *IEEE Trans. Parallel Distrib. Syst.*, 27(9):2533–2545, 2016. 118

[59] Thomas Grégoire and Adam Chlipala. Mostly automated formal verification of loop dependencies with applications to distributed stencil algorithms. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving (ITP)*, volume 9807 of *LNCS*, pages 167–183. Springer, 2016. 63

[60] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, 1988. 2

[61] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984. 82, 117

[62] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, August 2014. 81, 111

[63] 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)*, pages 1109–1113, May 2017. 98

[64] Derrick Harris. Facebook's trillion-edge, hadoop-based and open source graph-processing. August 2013. 81, 111

[65] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. *Frontiers of Computer Science*, 8(1):83–99, Feb 2014. 123, ix

[66] Charles A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. 7

[67] Charles A. R. Hoare. A note on the for statement. *BIT Numerical Mathematics*, 12(3):334–341, Sep 1972. 7

[68] Zhenjiang Hu, Hideya Iwasaki, and Masato Takechi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Trans. Program. Lang. Syst.*, 19(3):444–461, May 1997. 12, 21

[69] Zhenjiang Hu, Hideya Iwasaki, Masato Takechi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *ICFP*, 1997. 21

[70] Zhenjiang Hu, Masato Takeichi, and Hideya Iwasaki. Diffusion: Calculating efficient parallel programs. In Olivier Danvy, editor, *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1*, pages 85–94. University of Aarhus, 1999. 19, 98, 103

[71] Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997. 104

[72] Joab Jackson. Facebook's graph search puts Apache Giraph on the map. August 2013. 81, 111

[73] Jacobs, Bart. The Essence of Coq as a Formal System. `https://people.cs.kuleuven.be/~bart.jacobs/coq-essence.pdf`, April 2013. 31

[74] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *POPL 2015: 42nd symposium Principles of Programming Languages*, pages 247–259. ACM Press, 2015. 8

[75] Kazuhiko Kakehi, Kiminori Matsuzaki, and Kento Emoto. Efficient parallel tree reductions on distributed memory environments. *Scalable Computing: Practice and Experience*, 18(1):1–15, 2017. 51

[76] Arijit Khan. Vertex-centric graph processing: The good, the bad, and the ugly. *CoRR*, abs/1612.07404, 2016. 111

[77] Arijit Khan and Sameh Elnikety. Systems for big-graphs. *Proc. VLDB Endow.*, 7(13):1709–1710, August 2014. 111

[78] Daniel Kästner, Ulrich Wünsche, Jörg Barrho, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS 2018: Embedded Real Time Software and Systems*. SEE, January 2018. 8

[79] Palden Lama and Xiaobo Zhou. AROMA: Automated resource allocation and configuration of MapReduce environment in the cloud. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 63–72, New York, NY, USA, 2012. ACM. 81

[80] Ralf Lammel. Google's mapreduce programming model - revisited. *Science of Computer Programming*, 70(1):1 – 30, 2008. 79

[81] Joeffrey Légaux, Zhenjiang Hu, Frédéric Loulergue, Kiminori Matsuzaki, and Julien Tesson. Programming with BSP Homomorphisms. In *Euro-Par Parallel Processing*, number 8097 in LNCS, pages 446–457, Aachen, Germany, 2013. Springer. 79

[82] Joeffrey Légaux, Sylvain Jubertie, and Frédéric Loulergue. Development Effort and Performance Trade-off in High-Level Parallel Programming. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 162–169, Bologna, Italy, 2014. IEEE. 98

[83] Joeffrey Légaux, Frédéric Loulergue, and Sylvain Jubertie. Managing Arbitrary Distributions of Arrays in Orléans Skeleton Library. In *International Conf. on High Performance Computing and Simulation (HPCS)*, pages 437–444, Helsinki, Finland, 2013. IEEE. 64, 79

[84] Joeffrey Légaux, Frédéric Loulergue, and Sylvain Jubertie. OSL: an algorithmic skeleton library with exceptions. In *International Conference on Computational Science (ICCS)*, pages 260–269, Barcelona, Spain, 2013. Elsevier. 79

[85] Pierre Letouzey. Extraction in Coq: An overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms*, pages 359–369, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 41

[86] Mario Leyton and José M. Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. In *PDP*, pages 289–296. IEEE, 2010. 79

[87] R. Loogen, Y. Ortega-Mallen, and R. Pena-Mari. Parallel Functional Programming in Eden. *J Funct Program*, 3(15):431–475, 2005. 78

[88] Rita Loogen. Eden – Parallel Functional Programming with Haskell. In Viktória Zsòk, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School*, volume 7241 of *LNCS*, pages 142–206. Springer, 2012. 78

[89] Frédéric Loulergue. Implementing Algorithmic Skeletons with Bulk Synchronous Parallel ML. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 461–468. IEEE, 2017. 78

[90] Frédéric Loulergue. A verified accumulate algorithmic skeleton. In *Fifth International Symposium on Computing and Networking (CANDAR)*, pages 420–426, Aomori, Japan, November 19-22 2017. IEEE. 42

[91] Frédéric Loulergue, Wadoud Bousdira, Frédéric Gava, Louis Gesbert, Gaétan Hains, Guillaume Petiot, and Julien Tesson. Bulk Synchronous Parallel ML 0.5 Reference Manual. `https://traclifo.univ-orleans.fr/BSML/`, 2010. 28

[92] Frédéric Loulergue, Wadoud Bousdira, and Julien Tesson. Calculating Parallel Programs in Coq using List Homomorphisms. *International Journal of Parallel Programming*, 45(2):300–319, 2017. 8, 42

[93] Frédéric Loulergue, Frédéric Gava, and David Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In *International Conference on Computational Science (ICCS)*, volume 3515 of *LNCS*, pages 1046–1054. Springer, 2005. 28, 78

[94] Frédéric Loulergue, Simon Robillard, Julien Tesson, Joeffrey Légaux, and Zhenjiang Hu. Formal Derivation and Extraction of a Parallel Program for the All Nearest Smaller Values Problem. In *ACM Symposium on Applied Computing (SAC)*, pages 1577–1584, Gyeongju, Korea, 2014. ACM. 42

[95] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with i/o. *J. Symb. Comput.*, 46(2):95–118, 2011. 63

[96] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. 81, 111

[97] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *InfoScale'06: Proceedings of the 1st international conference on scalable information systems*. ACM, 2006. 64

[98] Kiminori Matsuzaki. Efficient Implementation of Tree Accumulations on Distributed-Memroy Parallel Computers. In *International Conference on Computational Science (ICCS)*, volume 4488 of *LNCS*. Springer, 2007. 72, 79

[99] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Implementation of parallel tree skeletons on distributed systems. In *The Third Asian*

Workshop on Programming Languages and Systems, APLAS'02, Shanghai Jiao Tong University, Shanghai, China, November 29 - December 1, 2002, Proceedings, pages 258–271, 2002. 51

[100] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallelization with tree skeletons. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*, volume 2790 of *Lecture Notes in Computer Science*, pages 789–798. Springer, 2003. 103

[101] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallel skeletons for manipulating general trees. *Parallel Computing*, 32(7-8):590–603, 2006. 76

[102] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Towards automatic parallelization of tree reductions in dynamic programming. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 39–48, New York, NY, USA, 2006. ACM. 74

[103] Jeanna Neefe Matthews and Thomas E. Anderson, editors. *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, 2009. 8

[104] William F. McColl. Lectures on parallel computation. chapter General Purpose Parallel Computing, pages 337–391. Cambridge University Press, New York, NY, USA, 1993. 27

[105] Lambert Meertens. Algorithmics – towards programming as a mathematical activity. In *Proceedings of CWI Symposium on Mathematics and Computer Science*, pages 289 – 334. North-Holland, 1986. 8

[106] Lambert Meertens. First steps towards the theory of rose trees. *Working paper 592 ROM-25*, 1988. 25

[107] Akimasa Morihata, Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Hideya Iwasaki. Optimizing declarative parallel distributed graph processing by using constraint solvers. In John P. Gallagher and Martin Sulzmann, editors, *Functional and Logic Programming*, pages 166–181, Cham, 2018. Springer International Publishing. 112

[108] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. The third homomorphism theorem on trees: Downward & upward lead to divide-and-conquer. *SIGPLAN Not.*, 44(1):177–185, January 2009. 98, 103

[109] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic inversion generates divide-and-conquer parallel programs. *SIGPLAN Not.*, 42(6):146–155, June 2007. 14, 21

[110] B. Nichols, D. Buttlar, and J. Farrell. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. A POSIX standard for better multiprocessing. O'Reilly Media, Incorporated, 1996. 4

[111] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag, Berlin, Heidelberg, 2007. 7

[112] Oliphant T. NumPy: numerical Python. `http://numpy.scipy.org`, 2010. 98

[113] Corey Pennycuff and Tim Weninger. Fast exact graph diameter computation with vertex programming. Barcelona Supercomputing Center, 2015. 111

[114] Ari Rasch and Sergei Gorlatch. Multi-dimensional homomorphisms and their implementation in OpenCL. *International Journal of Parallel Programming*, 46(1):101–119, Feb 2018. 81, 98, 109

[115] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, October 2015. 111

[116] Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computing Science*, pages 151–217, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. 81, 107

[117] Shigeyuki Sato and Kiminori Matsuzaki. A Generic Implementation of Tree Skeletons. *Int J Parallel Prog*, 44(3):686–707, 2016. 64, 79

[118] Patrick Schnider. An introduction to proof assistants. Student Seminar in Combinatorics: Mathematical Software, 2009. 8

[119] Julien Signoles, Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, and Boris Yakobowski. Frama-c - A software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012. 8, 30

[120] David Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, New York, NY, USA, 1995. 12, 111

[121] David Skillicorn. Parallel implementation of tree skeletons. *J. Parallel Distrib. Comput.*, 39(2):115–125, December 1996. 73

[122] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag. 38

[123] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13:134:1–134:25, April 2014. 78

[124] L.M. Surhone, M.T Timpledon, and S.F. Marseken. *Von Neumann Architecture: Central Processing Unit, John Von Neumann, Universal Turing Machine, SISD, Read-write Memory*. Betascript Publishing, 2010. 3

[125] Julien Tesson, Hideki Hashimoto, Zhenjiang Hu, Frédéric Loulergue, and Masato Takeichi. Program calculation in Coq. In Michael Johnson and Dusko Pavlovic, editors, *Algebraic Methodology and Software Technology*, pages 163–179, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. 8

[126] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`. 8, 30

[127] The Isabelle Development Team. Isabelle. `https://isabelle.in.tum.de/`. 30

[128] The OCaml Development Team. The OCaml system. `https://ocaml.org/index.fr.html`. 28

[129] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward 2013, pages 135–152, New York, NY, USA, 2013. ACM. 30

[130] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. *SIGPLAN Not.*, 49(6):530–541, June 2014. 30

[131] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990. 27

[132] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. 38

[133] Qi Wang, Meixian Chen, Yu Liu, and Zhenjiang Hu. Towards systematic parallel programming of graph problems via tree decomposition and tree parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 25–36, New York, NY, USA, 2013. ACM. 23

[134] Benjamin Welton, Evan Samanas, and Barton P. Miller. Mr. Scan: Extreme scale density-based clustering using a tree-based network of gpgpu nodes. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2013. 123, ix

[135] Chen Xiaoyun, Min Yufang, Zhao Yan, and Wang Ping. GMDBSCAN: Multi-density DBSCAN cluster based on grid. In *2008 IEEE International Conference on e-Business Engineering*, pages 780–783, Oct 2008. 82, 117

[136] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, and James Cheng. Big graph analytics systems. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2241–2243, New York, NY, USA, 2016. ACM. 111