Parallel Programming with OCaml: A Tutorial

Victor Allombert, Mathias Bourgoin, Frédéric Loulergue

Parallel Programming with OCaml: A Tutorial

Victor Allombert
Univ Orleans, INSA Centre Val-de-Loire, LIFO EA 4022
Orléans, France
victor.allombert@univ-orleans.fr

Mathias Bourgoin
Univ Orleans, INSA Centre Val-de-Loire, LIFO EA 4022
Orléans, France
mathias.bourgoin@univ-orleans.fr

Frédéric Loulergue
SICCS
Northern Arizona University
Flagstaf, USA
frederic.loulergue@nau.edu

INVITED TUTORIAL PAPER

***Abstract*—OCaml is a multi-paradigm (functional, imperative, object-oriented) high level sequential language. Types are statically inferred by the compiler and the type system is expressive and strong. These features make OCaml a very productive language for developing efficient and safe programs. In this tutorial we present three frameworks for using OCaml to program scalable parallel architectures: BSML, Multi-ML and Spoc.**

Keywords*— *Parallel programming; functional programming; bulk synchronous parallelism; Multi-BSP; GPGPU

As parallel architectures are now the norm, it is necessary to consider programming languages and libraries that offer a trade-off between execution efficiency and programming productivity. While languages and libraries for high performance computing, such as MPI [1], favor efficiency over productivity, a lot of application domains require that mainstream programmers develop parallel applications without huge knowledge about parallel programming.

Big Data is an area where productivity is also considered as important. MapReduce [2] is a well-known programming model where it is only possible to express a parallel algorithm in a very structured and specific way. The creators of MapReduce indicate functional programming as an inspiration for their framework. Moreover, there is a recent trend to include functional features in mainstream languages, such as Java or C++.

This tutorial is an introduction to parallel programming using a modern functional programming language: OCaml, a statically and strongly typed language, with a type inference system. This allows detecting errors very early in the development of applications. OCaml offers high-level features yet it is efficient. While the style of an OCaml application is mainly functional, OCaml is a multi paradigm language: it provides imperative and object-oriented features that can be used for efficiency or modularity reasons.

The three presented frameworks leverage OCaml for scalable parallel programming of homogeneous clusters (Section II), hierarchical architectures (Section III), and GPUs (Section IV). We use, as a running example, a parallel map and reduce implementations on distributed lists or distributed arrays. We begin with a short presentation of OCaml in Section I and discuss further reading in Section V.

I. An Overview of OCaml

In this section we describe the OCaml language and the features of the OCaml library that are necessary for a good understanding of this tutorial.

A. What is OCaml?

OCaml [3] is a functional programming language [4] from the ML (Meta Language) family. It is the main implementation of Caml, a general-purposes language developed at Inria.

OCaml relies on a powerful static type system allowing type inference, as well as parametric and generalized algebraic data types polymorphism. It also proposes a system of modules and an object oriented approach. The code generated by the compiler is thus safe, thanks to the type checker.

The OCaml language offers a bytecode compiler to allow code portability on various architectures. The quality of the native compilation allows performances close to the highest standards of modern compilers. OCaml also provides automatic memory management thanks to its embedded garbage collector. A *toplevel* that permits interactive use of OCaml through and interactive loop is also available.

OCaml is supported by a widespread community and has a rich set of libraries and development tools [5], [6]

B. Introduction to the OCaml Syntax

As OCaml provides type inference, it is not necessary to specify the type of a variable. Variables are defined using *let-bindings*. For example, `let x = 1` declares that `x` is a variable of type `int` with the value of `1`. Although unnecessary, it is possible to annotate an expression with a type, so `let x : int = 1` is valid too. We use type annotations in the next sections as types provide valuable information to the reader.

To declare a function in OCaml it is also possible to use a *let-binding* as following: `let prod x y = x *. y`. Here, we define the function `prod` which takes two arguments (`x` and `y`) and returns the product of them. Notice the usage of the `*.` operator which stands for floating point addition. The usage of this operators forces the types of `x` and `y` to be `float`. The `prod` function is typed `prod:float → float → float` as it takes as argument two floats and returns a float. The application of the function prod can be written as follows: `f 1.0 2.0`.

When there is no type constraints on values, the type system generates *forall types*. For example, `let f x = x` is a function taking an argument `x` and returning the exact same value. Here, the `f` can be applied on any value of `x`. Thus, it is typed `f : α → α` where α stands for any types.

Because of its functional capabilities, high-order functions are common in OCaml programs. For example, we can write a square function as following: `let square x = prod x x`. Anonymous function are also often used and are identified by the `fun` keyword. Thus, the following code allows the definition of the square function using anonymous function instead of using the `prod` function: `let square x = (fun x y → x *. y) x x`.

Lists are one of the most common data structure used in the context of functional programming. In Ocaml, a list is implemented as an immutable singly linked list. A list of integers in OCaml can be defined as following: `let l1 = [1;2;3]`. The type of the `l1` is `int list`. The brackets delimits the list and elements are separated with the semicolon. The empty list is written `[]` and the concatenation of two lists is written `l1 @ [4;5;6]`. The infix operator `::` adds an elements at the beginning of a list. Therefore `l1` can also be written `1::2::3::[]`. Actually `[]` and `::` are the two list *constructors*, i.e. any list value can be described in terms of this constant and this operation.

The definition of recursive functions using the let-binding must be annotated by the `rec` keyword. For example, we can write a (non tail-)recursive function that builds a list containing the values from `n1` to `n2` as follows:

```
let rec from_to n1 n2 =
  if n1>n2
  then []
  else n1::(from_to (n1+1) n2)
```

This function returns `[]` if the lower bound is greater than the upper bound. Otherwise, it returns the value of `n1` concatenated with the recursive call of `from_to` on `n1+1` and `n2` using the operator `::`.

Pattern matching is a powerfully control structure used to match values or expressions (patterns) to computations. Clauses are matched, in order, and the first expression corresponding to the matching clause is evaluated. To describe how pattern matching works, we implement a (non tail-recursive) map operator in the following example. We recall that map on lists applies a function on each elements of a given list.

```
let rec map f l =
  match l with
  | [] → []
  | hd::tl → (f hd)::(map f tl)
```

As expected, `map` takes as argument a function `f` and a list of elements `l`. The pattern matching is expressed using the `match with` keywords. Here, we *match* the value of `l` *with*, respectively:

- `[]` is the empty list. We return an empty list if the input list is empty. Indeed, if the previous matching fails, it means that the list is not composed of at least two elements, thus it is an empty list.
- `hd::tl` a list composed of the head element `hd` and the tail `tl` (the rest of the list). In this case, we return the result of the application of `f` on `h` concatenated with the result of the recursive call on the tail.

It is also possible to define arrays in OCaml as follows: `let a = [|1;2;3|]`. Thus, `a` is typed `int array`. Unlike lists, arrays are mutable data structures. Updating the cell at index `i` of an array `a` with a value `v` is written `a.(i)←v`.

In OCaml, it is simple to define its own types using records. A record is a data structure holding multiple values. For example, the definition of a point of a Cartesian coordinate system can be written:

```
type point = {
  name : string;
  x : float;
  y : float;
}
```

Here, the record `point` is made of three fields: the name of the point and the x and y coordinates. An instance of a point is written: `let a = {name="A";x=3.0;y=5.0}`. Records can have `mutable` fields that can be set using the ← operator. For example, annotating the field `x` as follows `mutable x : float` allows one to write `a.x←4.0`.

C. Some Features of OCaml Libraries

To manipulate common data structures such as lists or arrays it is possible to use the `List` and `Array` modules from the standard library.

Regarding lists, the `map` function is very useful to apply a function on each element of a list. `List.map` is typed `map : (α→β)→α list→β list`. The code `List.map f l` applies function `f` on each elements of the list `l`. Thus, `List.map (fun x → x*x) [0;1;2;3]` produces list `[0;1;4;9]`.

Folding on lists is a common operation that can be done using `List.fold_left : (α→β→α)→α→β list→α`. Given a function `f`, an element `e` and a list of n elements `[x0;x1; ... ;xn]`, the informal semantics of `List.fold_left` is: `f (... (f (f e x0) x1) ...) xn`. To compute the sum of all the element of a list, we can write the following code: `List.fold_left (fun x y → x + y) 0 [1;2;3;4;5]`.

Many other functions are available to get the size (`List.length`), access the n^{th} element (List.nth), reverse (`List.rev`), and so on.

A similar set of functions is available for arrays.

II. Bulk Synchronous Parallelism with OCaml

Bulk Synchronous Parallel ML or BSML [7] is based on the Bulk Synchronous Parallel model [8], [9], [10] or BSP. The aim of this model is to be a bridge between the world of parallel architectures and the world of parallel applications.

A. The Bulk Synchronous Parallel Model

The BSP model assumes a BSP architecture: a parallel machine consisting of a set of homogeneous processor-memory pairs, a network delivering point-to-point communications, and a global synchronization unit. This BSP architecture can be mapped on any general purpose parallel architecture. For example, a cluster can be seen as a BSP machine even if in this case the global synchronization unit is usually implemented using software rather than hardware. In a shared memory machine, each core can work on a dedicated region of the shared memory and can communicate with other cores using another region of the shared memory.

The execution of a BSP program proceeds as a sequence of *super-steps*. A super-step starts with a pure computation phase where each processor computes using the data it holds in its private memory, and requests data from other processors. A communication phase then occurs: data is exchanged between processors. Finally the super-step ends with a synchronization barrier. The remote data requested by processors is only guaranteed to be delivered after the synchronization barrier ends. The computation can then continue with the next super-step.

This constrained form of parallel execution made the design of a simple BSP performance model possible. A BSP architecture is characterized by four parameters: p is the number of processor-memory pairs ($\mathbb{P}$ denotes the set of processor identifiers from 0 to $p-1$), r is a processors' computing power usually expressed in floating point operations per second, L is the time required to perform a global synchronization barrier, and g (expressed in seconds per word) is the time required for performing a communication phase where each processor sends or receives at most 1 word of data. When the communication phase consists of the exchange of at most h words of data for each processor, the execution requires $h \times g$. Such a communication phase is said to realize a communication pattern named a h-relation. In the presentation of BSP algorithms, L and g are usually normalized with respect to r and are therefore respectively expressed in flops and flops per word.

If w_i is the time (or flops) required by processor i to execute the computation phase of the super-step, and the communication pattern of the super-step is a h-relation then the overall cost of the super-step is:

$$\max_{i \in \mathbb{P}} w_i + h \times g + L$$

The cost of the execution of a BSP program is the sum of the costs of its super-steps.

B. BSML Primitives

Currently BSML is implemented as a library for the OCaml language rather than a new language. The only difference is that a full language would provide a dedicated type system [11].

BSML implementation provides: an interactive loop to ease the development of BSML programs, a sequential implementation (used in the interactive loop), and a parallel implementation on top of MPI [7]. There also exists an implementation as an interactive web site[1].

BSML provides four constants corresponding to the four BSP parameters: **bsp_p**, **bsp_r**, **bsp_g** and **bsp_l**. This allows one to write performance portable programs: depending on the BSP parameters, it is possible to choose the best algorithm for the specific underlying architecture running the program.

BSML relies on a dedicated data structure, called *parallel vector*. This is a polymorphic data structure and its type is α `par`. The size of a parallel vector is always **bsp_p**, and each processor holds a value of type α. There is a limitation to polymorphism: α cannot be instantiate to a type that contains a parallel vector type. The type system [11] can statically reject such forbidden nesting. However, the current implementation does not provide this type system, it is therefore the responsibility of the programmer to avoid such nesting.

BSML has a pure functional semantics [12]. Therefore the sequential implementation and the parallel implementation give the same results, provided no imperative features of OCaml are used. If imperative features are incorrectly used, the results may differ. The type system can also statically detect such incorrect programs. One important advantage of such a semantics, is that it is possible to use a proof assistant to verify the correctness of BSML programs.

BSML is more concise than the standard BSPlib [13]. In addition to the four constants, it only provides four parallel functions to manipulate parallel vectors.

In the remainder of this section, $\langle v_0, \ \ldots, \ v_{p-1} \rangle$ denotes a parallel vector such that value v_i is in the local memory of processor i. In BSML, parallel vectors are manipulated globally: there is no index notation for accessing a specific element in a parallel vector.

The primitive **mkpar** is devoted to create parallel vectors. Its type is $(\texttt{int} \to \alpha) \to \alpha$ `par`. From a function f it creates a parallel vector: each processor applies this function to its identifier to obtain a value. Therefore **mkpar** $f = \langle f\ 0, \ \ldots, \ f\ (p-1) \rangle$.

A parallel vector can contain any type of element but a parallel vector. In particular it can contain functions. However, a parallel vector of functions is not a function. The primitive **apply** is used to apply a parallel vector of functions to a parallel vector of values. Its type is $(\alpha \to \beta)$ `par` $\to \alpha$ `par` $\to \beta$ `par`. We have **apply** $\langle f_0, \ \ldots, \ f_{p-1} \rangle \ \langle v_0, \ \ldots, \ v_{p-1} \rangle$
$= \langle f_0\ v_0, \ \ldots, \ f_{p-1}\ v_{p-1} \rangle$.

Both **mkpar** and **apply** are executed during the local computation phase of a BSP super-step.

An almost inverse of **mkpar** is **proj**. From a parallel vector it creates a function such that if it is applied to a processor identifier it will return the value held by this processor. **proj** has type α `par` $\to (\texttt{int} \to \alpha)$. **proj** is not completely an inverse of **mkpar** because even if a function `f` given as argument to **mkpar** is defined for integers that are not valid processor

[1] http://tesson.julien.free.fr/try-bsml

identifiers, **proj**(**mkpar** f) will only be defined for valid processor identifiers. **proj** requires a full super-step for its execution: each processor sends the value it holds to all the other processors, and a synchronization barrier is called.

Finally **put** can realize any h-relation of a communication phase. Its type is **put**:(int→α)par→(int→α)par. Therefore its input and output are parallel vectors of functions. In the input, each function in the parallel vector encodes the messages to be sent to other processors. For example, a function f_i at processor i encodes the messages to be sent by i to other processor. More precisely the message to be sent to processor j is $f_i\ j$. In the output, each function encodes the messages received by the processor that holds the function. For example, at processor j, function g_j is such that $g_j\ i$ is the message received by processor j from processor i. As described above, **put** seems to perform a total exchange. It is however not the case. Some values are considered to represent the empty message. This is for example the case for the empty list [] and the None value of the option type. A call to **put** needs a full super-step for its execution.

C. Map and Reduce with BSML

In this section, we consider that a distributed list is implemented as a parallel vector of lists. Intuitively, the user of the library we implement here can think of this data structure as a sequential list obtained by the concatenation of the lists held locally by p processors.

Implementing a parallel map is then very easy:

```
let map (f:α→β) (dl:(α list)par): (β list)par=
  apply (replicate (List.map f)) dl
```

First we replicate function f, meaning each processor will contain a copy of f, then we use **apply** to apply f to the local list on each processor. Replication can be defined as:

```
let replicate (x:α) : α par =
  mkpar(fun pid→x)
```

This is very common to **apply** a replicated sequential function, and we define a function to do that:

```
let parfun (f:α→β)(v: α par) : β par =
  apply(replicate f) v
```

Parallel reduction can be implemented in many ways. We present here two different implementations both using only one super-step of execution. reduce takes three arguments: a binary operation op, a neutral e for this binary operation, and a distribution list dl, and it reduces the distributed list using this binary operation. For this parallel function to compute the same result as a sequential reduction (implemented as locred below), op should be associative. The first implementation is based on **proj**:

```
let reduce(op:α→α→α)(e:α)(dl:(α list)par):α=
  let locred  = List.fold_left op e in
  let partial = parfun locred dl in
  locred (to_list partial)
```

We first perform local reductions on each processor obtaining a parallel vector of partial reductions named partial. Then using to_list this parallel vector is converted into a sequential list that is finally reduced. to_list can be implemented using **proj**:

```
let to_list (dl: α par) : α list =
  let f = proj dl in
  List.map f (from_to 0 (bsp_p−1))
```

If we prefer to obtain a parallel vector that contains the reduction result everywhere, reduce can be implemented using **put** as shown in Figure 1.

```
let reduce op e dl : α par =
  let locred  = List.fold_left op e in
  let partial = parfun locred dl in
  let mkmsg   = fun data dst → data in
  let totex   = put(parfun mkmsg partial) in
  let procs   = from_to 0 (bsp_p−1) in
  parfun(fun f→locred(List.map f procs))totex
```

Fig. 1. Parallel Reduction in BSML

```
let reduce op e dl : α par =
  let locred  = List.fold_left op e in
  let partial = ≪ locred $dl$ ≫ in
  let totex   = put ≪ fun dst → $partial$ ≫ in
  let procs   = from_to 0 (bsp_p−1) in
  ≪ locred (List.map $totex$ procs) ≫
```

Fig. 2. Parallel Reduction in BSML (New Syntax)

An alternative syntax avoids the use of **mkpar** and **apply**. It is based on this defined as **mkpar**(**fun** pid→pid), and a more compact syntax for manipulating parallel vectors [14].

Instead of writing replicate x it is possible to write ≪ x ≫. If a value v has type α par then inside ≪ and ≫ is is possible to use the notation v that has type α but has a different value on each processor: the value held by this processor in parallel vector v. For example the following expression: ≪(**fun** x → x+1) $this$≫ evaluates to vector $\langle 1, \ldots, p \rangle$.

The example of Figure 1 can be rewritten using the alternative syntax as shown in Figure 2.

III. Hierarchical Parallelism with OCaml

Multi-ML [15], [16] is a parallel programming language based on the Multi-BSP model [17]. It aims to bridge hierarchical architectures with several levels of distinct memories to parallel programming abstractions.

A. The Multi-BSP Model

Multi-BSP [17] is a structured parallel programming model which extends the BSP model [8]. It takes into account hierarchical architectures with an arbitrary number of levels. In accordance to the BSP model, it reflects the physical characteristics of the architecture. The structure and abstraction brought by Multi-BSP allows having portable programs with scalable performance predictions, without dealing with low-level details of the architectures. Note that the definition of

the Multi-BSP model handles balanced and homogeneous architecture only.

Multi-BSP brings a tree-based view of nested components where the lowest stages (leaves) are processors and every other stage (nodes) contains memory. The architecture depth is denoted by d and a Multi-BSP instance is defined by $4d$ parameters : (p_i, g_i, L_i, m_i). Where p_i is the number of sub-components inside the $i-1$ stage; g_i is the bandwidth between stages i and $i-1$; L_i is the synchronisation cost of all sub-components of a component of $i-1$ stage; m_i is the amount of memory available at stage i for each component of this stage.

Every component of the architecture can execute independent codes. However, they have to synchronise in favour of data exchange. Thus, the Multi-BSP model does not allow subgroup synchronisation of any group of processors. The execution scheme of Multi-BSP algorithms is similar to the BSP one. Here, supersteps are hierarchically organised in order to manage sub-synchronisations. At a stage i, a superstep is composed by the following steps: the sub-components of stage $i-1$ execute some code independently (until they reach a barrier); there are exchanges of information with the m_i memory; synchronisation of the sub-components (stage $i-1$).

Thanks to the structure of Multi-BSP and a dedicated cost model, performance prediction is also possible. Considering the communication cost C_j^i of a superstep j at stage i as follows:

$$C_j^i = h_j \times \mathbf{g}_i + \mathbf{L}_i$$

where h_j is the maximum size of the exchanged messages at superstep (as it is in the BSP cost model). We can express the cost T of a Multi-BSP algorithms as follows:

$$T = \sum_{i=0}^{d-1} \left(\sum_{j=0}^{N_i-1} w_j^i + C_j^i \right)$$

where d is the depth of the Multi-BSP architecture, N_i is the number of supersteps at stage i, w_j^i is the maximum computational cost of the superstep j within stage i.

B. The Multi-ML Language

Multi-ML [15], [16] is based on the idea of executing BSML-like codes on every stage of a Multi-BSP architecture. This approach eases *incremental* development from BSML codes to Multi-ML ones. As expected, the synchronous communication primitives of BSML are also available to communicate values from/to parallel vectors. Multi-ML follows the Multi-BSP approach where the hierarchical architecture is composed by *nodes* and *leaves*. On nodes, it is possible to build parallel vectors, like in BSML. A parallel vector aims to manage values that are stored on the sub-nodes: at stage i, the code **`let`** `v=≪ e ≫` evaluates the expression `e` on all $i-1$ stages.

Multi-ML also introduces the concept of *multi-function* to recursively go through a Multi-BSP architecture. A *multi-function* is a particular recursive function, defined by the keyword **`let multi`**, which is composed by two codes: the node and the leaf codes. The *recursion* is initiated by calling the multi-function (recursively) inside the scope of a parallel vector, that is to say, on the sub-nodes. The evaluation of a multi-function starts on the root node and is propagated down to the leaves. Then, the recursion goes back to the root node. The execution of a multi-function can be seen as a *divide and conquer* pattern. Another parallel data structure called *tree* is available in Multi-ML. A tree denotes a value which is distributed over all components of the Multi-BSP architecture. It is possible, for each component, to access its own value using **`at`** `t`, where `t` is a tree. The keyword **`let multi tree`** is used to define a *multi-tree-function* allowing tree construction. A multi-tree-function behaves similarly to a multi-function, but the keyword **`finally`** must be used to specify the values used to build the tree. Thus, at each node, we specify the local root and the branches used to build the *sub-tree* of the current stage; and so on, toward the root node.

C. Map and Reduce with Multi-ML

As the Multi-ML language uses a hierarchical tree based recursion to execute code, the map and reduce phases are almost embedded in the syntax of the language. Indeed, the map phase consists in mapping a function to the leaves (the map phase); and then, we combine the results to the root node (the reduce phase).

Similarly to BSML, we consider that a distributed list is implemented as a tree where only the leaves contain values (and nodes contain empty lists). The `m_map` function (standing for *multi-map*) can be written as following:

```
let m_map(f:α→β)(tdl:α list tree) =
  let multi tree map tdl =
    where node =
      let rc = ≪  map tdl ≫  in
      finally(rc,[])
    where leaf =
      List.map f (at tdl)
  in map tdl
```

First, the `map` multi-function is enclosed into the `m_map` function in order to define the function `f` globally and avoid its communications through each recursive call. As expected, the **`let multi tree`** keyword is used to declare `map` as its executions results in a tree. Then, we define the **`node`** code by the recursive call of `map` on its sub-nodes. The result of the recursion is stored in `rc` and will contains the sub-trees (or branches) generated on the sub-nodes. Thus, the node sections ends by constructing a new sub-tree with lower branches and an empty list. Finally, the leaf code consists of the sequential mapping of `f` on the values contained on leaves, accessed using **`at`** on `tdl`.

During the reduce phase, we must take into account the potential different number of children of a node when converting a parallel vector to a list (`to_list` function). To do so, we use the `nb_children` primitive which computes the number of children when executed on a node. The function `to_list` is thus defined:

```
let to_list vec =
  let p = proj vec in
  List.map p (my_children())
```

The reduce phase can be written easily:

```
let m_reduce(op:α→α→α)(e:α)(tdl:α list tree)=
  let multi reduce tdl =
    where node =
      let rc = ≪  reduce tdl ≫  in
      let sub_vals = to_list rc in
      List.fold_left op e sub_vals
    where leaf =
      List.fold_left op e (at tdl)
in reduce tdl
```

Using the reduce operator `op` and a neutral element `e`, the node code of the `reduce` multi-function consists in calling `reduce` recursively. The resulting parallel vector is then converted into a list using `to_list`. The resulting value of a node is thus the local reduction of the sub-reductions. The leaf code is the local reduction of the values distributed through the `tdl` tree. As `reduce` is a multi-function, it returns a sequential value (typed β) which is then available to all the components of the architecture.

It is also possible to write a single multi-function combining both the map and reduce phase. Such a multi-function will benefit in terms of line of code and efficiency, as the recursion through the components is done once. The `m_map_reduce` function can be written as follows:

```
let m_map_reduce (op_map:α→β) (op_red:β→β→β)
                 (e:β) (tdl:α list tree):β =
  let multi map_reduce tdl =
    where node =
      let rc = ≪  map_reduce tdl ≫  in
      let sub_vals = to_list rc in
      List.fold_left op_red e sub_vals
    where leaf =
      let res_map = List.map op_map (at tdl) in
      List.fold_left op_red e res_map
in map_reduce tdl
```

Intuitively, the node code combines the recursion call of `map_reduce` and the local reduction. The leaf performs both the mapping phase and the local reduction.

IV. GPGPU Programming with OCaml

A. GPGPU Programming

General purpose programming with graphics processing unit (GPGPU) is a common way to achieve high performance. It consists in the use of highly parallel GPU architectures as co-processors to handle intensive computations. GPGPU programming demands to write specific subprograms (called kernels) to be actually executed on the GPUs. As GPUs are co-processors, kernels are handled via the CPU host. The CPU is also responsible for the memory management (Direct Memory Access (DMA) copy between the host memory and the GPU memory). GPUs are parallel architectures (several hundreds of "cores") and thus kernels are written with a specific programming style.

B. The SPOC Library and the Sarek Domain Specific Language

SPOC (Stream Processing with OCaml) [18], [19] is a high-level library for GPGPU programming with OCaml. SPOC is a library based on programming *via* the OpenCL and CUDA frameworks. It offers several abstractions over memory transfers as well as device management. SPOC only focuses on the host program. It can interoperate with native GPGPU kernels (using the OpenCL and CUDA C subsets) as well as with kernels described with the Sarek DSL. To provide portability, SPOC unifies both CUDA and OpenCL APIs. SPOC automatically detects, at run-time, all devices compatible with it. As SPOC exposes a common API, it can be used to indifferently and conjointly handle multiple co-processors (from any framework). This eases the expression of complex programs dedicated to very heterogeneous architectures. SPOC introduces a specific data set to OCaml that is called `vector`. Vectors keep information about their current location in the system on host or co-processor memory. Thus, SPOC can automatically trigger transfers when needed. In particular, SPOC checks that every vector used by a GPGPU kernel is present in the co-processor memory (and triggers transfers if required) before launching the computation. Similarly, when the host reads or writes in a vector, SPOC checks its location and transfers it if needed.

SPOC can be used with native kernels as well as with kernels described in the Sarek DSL. In this section, we will only focus on Sarek to program kernels. Sarek is built into OCaml. As such, it provides some consistency over the host language. It is an expression oriented language with an ML-like imperative core. Sarek is based on the C subsets of CUDA and OpenCL but offers type inference, with static type checking as well as an OCaml-like syntax.

Using SPOC and Sarek, the common example of the vector addition can be written as following:

```
open Spoc
let vec_add = kern a b c n →
  let open Std in
  let idx = thread_idx_x +
            block_dim_x * block_idx_x  in
  if idx < n then
    c.[<idx>] ← a.[<idx>] +. b.[<idx>]

let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let block = {blockX = 1024;
             blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024;
          gridY=1; gridZ=1}

let () =
  random_fill v1;
  random_fill v2;
  Kirc.gen vec_add;
  Kirc.run vec_add  (v1, v2, v3, n)
               (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f; " i v3.[<i>]
  done;
```

The first function, `vec_add`, is the GPGPU kernel. As Sarek is built as an OCaml syntax extension, the kernel is written with Sarek directly into the OCaml code. As with CUDA or OpenCL, a kernel is an elementary computation that will be instantiated into GPGPU-threads. These threads will then be mapped over the numerous hardware computation units of the GPU to execute the computation in parallel. Here, our kernel first opens the `Std` module that gives access to GPGPU-specific global functions, in particular those necessary to identify each thread running the kernel. Here we use `thread_idx_x + block_dim_x * block_idx_x` to compute the global identifier of the each thread running an instance of our kernel. Then our kernel naively computes an elementary addition. The mapping of this kernel on numerous computation units will provide the overall vector addition.

Following the kernel comes the host part of the program, that is run by the CPU and will handle communications and computations scheduling with the GPU. It starts by initializing the SPOC system with `Devices.init()`. This function scans the system for co-processors compatible with either CUDA or OpenCL and returns an array of devices that can be used in the rest of the program. We then declare some variables. In particular, `v1, v2` and `v3` that are SPOC vectors, that can be shared between host and GPU automatically. The `block` and `grid` variables are used to describe a 3D layout of threads corresponding to a *virtual GPU* appropriately sized for our computation. With GPGPU frameworks, kernels are executed in a 3D grid of 3D blocks of threads. Threads will be mapped to the actual computation units of the GPU. Threads inside a block can be synchronized and can communicate through a shared memory. There is no global synchronization mechanism (between threads of different blocks). Here we describe a linear layout with one thread for each element in our vectors. Then, we have the main function, it first fills the vectors with random values. Then we use the `gen` function from the `Kirc` module to compile our kernel to actual CUDA or OpenCL code. Here we use the default compilation scheme that generates both versions of the kernel. Then we run the kernel with the correct parameters on the first device detected during initialization (`dev.(0)`). Finally the program prints the content of the result vector that was computed on the GPU. As described earlier, the host program does not need to explicitly handle memory transfers between the host and the GPU as they are automatically managed by the SPOC library. Besides, remember that this simple example is very portable as compiled program using SPOC and Sarek are usable with CUDA or OpenCL devices (or a combination of both) specifying their behaviour at run-time.

C. Map and Reduce with SPOC and Sarek

To make GPGPU programming simpler, both CUDA and OpenCL frameworks impose specific constraints. SPOC and Sarek inherited some of them which makes writing general Map and Reduce functions trickier than with BSML and Multi-ML. In particular, kernel functions are sequential (it is their mapping on numerous computations units that produces the parallelism). Besides, higher order functions are not available in the Sarek DSL, which makes it impossible to write parallel map/reduce-like functions with Sarek only. On the other hand, OCaml (and the SPOC library) and Sarek look very similar but, in practice, Sarek code is very different from OCaml functions. Sarek kernels are pre-compiled into a data-structure containing an internal representation (IR) of the kernel that is directly embedded into the OCaml host code at compile-time. Then the `Kirc.gen` function we've seen earlier will translate this IR into actual CUDA/OpenCL code. This make difficult (but not impossible) to write a generalized map/reduce operation from OCaml with SPOC that manipulated Sarek kernels. Last but not least, Sarek kernels cannot currently be polymorphic. Keeping all these constraints in mind, we will, in the rest of this section, look at how to write map/reduce-like computations with Sarek, SPOC and OCaml.

1) Map: The main objective here is to use a Sarek kernel producing an output vector of type β from an input vector of type α. A naive Sarek kernel would thus look like the following:

```
let map_f = kern a b →
let open Std in
let idx = thread_idx_x +
          block_dim_x * block_idx_x in
if idx < Vector.length a then
b.[<idx>] ← f a.[<idx>]
```

Here `f` is an external function or an inlined computation of type $\alpha \rightarrow \beta$.

From the host side, it is necessary to pre-create a vector for the output of the map computation before running the kernel. The code would then look like the following :

```
let dev = Devices.init ()

(* α represents the vector elements type *)
(* n is the size of the vector *)
let input = Vector.create α n

(* populate the inpute vector *)
...

(* prepare the launch of the map kernel*)
let output = Vector.create β n

(* blockX could be any multiple of 32 threads
   depending on n, specific optimizations or
   actual hardware *)
let block = {blockX = 1024;
  blockY = 1; blockZ = 1}
(* gridX is computed in order to use enough
   blocks to have one thread per element *)
let grid={gridX=(n+block.blockX-1)/block.blockX;
          gridY=1; gridZ=1};

(* generate and run the map_f kernel *)
Kirc.gen map_f;
Kirc.run map_f (input, output) (block,grid)
              dev.(0);
(* from now, output contains the result of the
   application of f on each element of input *)
```

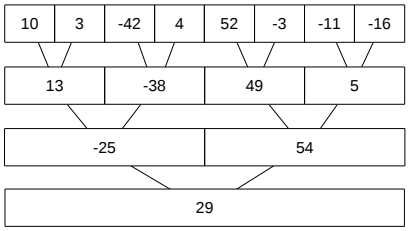

Fig. 3. A tree based reduction using the addition operator.

Using this kind of program, the application of the `f` function (used in the Sarek kernel) to each elements of the input vector into the output vector will take place in parallel using a GPU. The version proposed here is naive as it does not take into account the arithmetic complexity of the `f` function which could be too small to dedicate one GPU thread to each element of the vector. In the same vein, it cannot be used directly with vectors larger than the maximum number of threads usable on an actual GPU.

2) Reduce: Map is an easy target for parallelism. Reduction is more complex. The common/simple way to handle reduction with GPUs is a tree based approach. A tree based GPU computation of a reduction using the addition operator on the vector would be similar to Figure. 3.

Here, each level of the tree corresponds to a parallel computation. Between each computation, a synchronization is needed. However, GPUs cannot do global synchronizations inside kernels. The only synchronization mechanism available synchronizes threads within a block. There is no similar mechanism between blocks. Blocks are limited in size (number of threads), besides actual hardware demands to use multiple blocks in order to increase parallelism. Thus, for large enough vectors, global synchronization (between blocks) will be needed. The simpler solution to provide "global synchronization" consists in decomposing the overall reduction into multiple kernels (one for each step of the reduction), using kernel launches as global synchronization points. We will not detail here the code for a GPGPU reduction, CUDA/OpenCL reductions have been widely discussed [20], [21] and is similar with Sarek and OCaml.

3) High-level Map/Reduce with SPOC and Sarek: As we said earlier, Map and Reduce with SPOC and Sarek are difficult to generalize. In the end, the two solutions proposed here are very low-level and similar to those available with CUDA and OpenCL. However, without going into detail, Sarek offers a solution to improve the situation. Sarek kernels internal representation is embedded into the OCaml host code at compile-time. This IR is mostly an Abstract Data Tree (ADT) compose of OCaml values. This makes this tree parsable from the host code. For instance, the `Kirc.gen` function is an OCaml function parsing the IR ADT and generating CUDA/OpenCL native code. It is thus possible to design OCaml functions taking Sarek kernels as parameters and producing new computations. These computations (that we call transformations) can be used to provide map/reduce-like operations. For instance, for map, it consists mainly in an OCaml function taking a kernel computing on scalars (corresponding to the function `f` in the `map_f` kernel) as a parameter. The function transforms it into a new kernel similar to `map_f`. It also generates the output vector and runs the kernel with the correct input/output before returning the computed output vector.

For instance, raising each element of a vector to the power of two could be written as follow :

```
let f = kern a → a * a in
...
let output = map f input dev.(0)
```

The `f` function is still constrained by the limitations of GPGPU frameworks (no polymorphism etc.). However, the map transformation is polymorphic and can transform any kernel of type $\alpha \rightarrow \beta$.

V. Further Reading

a) BSML: While being more concise than BSPlib, BSML is however equally expressive [22]. In addition to the primitives presented in this paper, BSML has been extended by parallel compositions [23], [24], [25], [26], parallel exceptions handling [27], and IO. These features are not available in the public release yet.

The pure semantics of BSML makes the verification of BSML program possible using a proof assistant [28]. This is the foundation of a framework for the Coq proof assistant to support the systematic development of correct parallel programs [29], [30], [31], [32], [33].

BSML has been used for example to implement parallel data structures [34], algorithmic skeletons on distributed lists [35], and powerlists [36].

b) Multi-ML: In the tradition of BSML, Multi-ML aims to propose formal properties. The semantics of the language is thus described in [37] and a formalisation of the compilation scheme ensuring that the generated code follows the semantics is proposed in [38]. As the language relies on a type system with locality annotations and effects, a formal definition the Multi-ML type system can be found in [15].

c) SPOC: The SPOC library and the Sarek DSL have been first described in [18] and [19]. Several extensions have been developed on top of them. Parallel skeletons and compositions have been proposed in [39], [40] to make the design of high performance GPGPU applications easier. High-level data structures that can automatically be transferred between host (CPU) and GPU memory have been introduced to Sarek in [41]. A portable profiling solution has been developed as a Sarek extension in [42]. They also have been used to develop a web-oriented version of SPOC described in [43].

References

[1] M. Snir and W. Gropp, *MPI the Complete Reference*. MIT Press, 1998.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*. USENIX Association, 2004, pp. 137–150. [Online]. Available: http://static.usenix.org/events/osdi04/tech/full_papers/dean/dean.pdf

[3] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, "The OCaml System release 4.00.0," http://caml.inria.fr, 2012.

[4] G. Cousineau and M. Mauny, *The Functional Approach to Programming*. Cambridge University Press, 1998.

[5] E. Chailloux, P. Manoury, and B. Pagano, *Développement d'applications avec Objective Caml*. O'Reilly France, 2000, freely available in english at http://caml.inria.fr/oreilly-book.

[6] Y. Minsky, "OCaml for the masses," *Commun. ACM*, vol. 54, no. 11, pp. 53–58, 2011.

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

[8] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, p. 103, 1990.

[9] W. F. McColl, "Scalability, portability and predictability: The BSP approach to parallel programming," *Future Generation Computer Systems*, vol. 12, pp. 265–272, 1996.

[10] R. H. Bisseling, *Parallel Scientific Computation*. Oxford University Press, 2004.

[11] F. Gava and F. Loulergue, "A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting," *Future Gener Comp Sy*, vol. 21, no. 5, pp. 665–671, 2005.

[12] F. Loulergue, G. Hains, and C. Foisy, "A Calculus of Functional BSP Programs," vol. 37, no. 1-3, pp. 253–277, 2000.

[13] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling, "BSPlib: The BSP Programming Library," *Parallel Computing*, vol. 24, pp. 1947–1980, 1998.

[14] W. Bousdira, F. Gava, L. Gesbert, F. Loulergue, and G. Petiot, "Functional Parallel Programming with Revised Bulk Synchronous Parallel ML," in *First International Conference on Networking and Computing (ICNC 2010), 2nd International Workshop on Parallel and Distributed Algorithms and Applications (PDAA)*, K. Nakano, Ed. IEEE Computer Society, 2010, pp. 191–196.

[15] V. Allombert, "Functional Abstraction for Programming Multi-Level Architectures: Formalisation and Implementation," Ph.D. dissertation, Université Paris Est, Créteil, France, Jul. 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01693568

[16] V. Allombert, F. Gava, and J. Tesson, "Multi-ML: Programming Multi-BSP Algorithms in ML," *International Journal of Parallel Programming*, vol. 45(2), p. 20, 2017. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01160164

[17] L. G. Valiant, "A Bridging Model for Multi-core Computing," *J. Comput. Syst. Sci.*, vol. 77, no. 1, pp. 154–166, Jan. 2011.

[18] Bourgoin, Mathias and Chailloux, Emmanuel and Lamotte, Jean-Luc, "Spoc: Gpgpu programming through stream processing with ocaml," *Parallel Processing Letters*, vol. 22, no. 02, p. 1240007, 2012.

[19] ——, "Efficient abstractions for GPGPU programming," *International Journal of Parallel Programming*, vol. 42, no. 4, pp. 583–600, 2014.

[20] Harris, Mark, "Optimizing CUDA," 2007.

[21] Catanzaro, Bryan, "OpenCL Optimization Case Study: Simple Reductions," 2010.

[22] F. Loulergue, "A BSPlib-style API for Bulk Synchronous Parallel ML," *Scalable Computing: Practice and Experience*, vol. 18, pp. 261–274, 2017.

[23] ——, "Parallel Superposition for Bulk Synchronous Parallel ML," in *International Conference on Computational Science (ICCS)*, ser. LNCS, no. 2659. Springer Verlag, 2003, pp. 223–232.

[24] ——, "Parallel Juxtaposition for Bulk Synchronous Parallel ML," in *Euro-Par 2003*, ser. LNCS, H. Kosch, L. Boszormányi, and H. Hellwagner, Eds., no. 2790. Springer Verlag, 2003, pp. 781–788.

[25] F. Gava, "Implementation of the parallel superposition in bulk-synchronous parallel ML," in *7th International Conference on Computational Science (ICCS), Beijing, China, May 27-30*, ser. LNCS, vol. 4487. Springer, 2007, pp. 611–619.

[26] I. Garnier and F. Gava, "CPS implementation of a BSP composition primitive with application to the implementation of algorithmic skeletons," *I. J. Parallel, Emergent and Distributed Systems*, vol. 26, no. 4, pp. 251–273, 2011.

[27] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski, "Bulk Synchronous Parallel ML with Exceptions," *Future Gener Comp Sy*, vol. 26, no. 3, pp. 486–490, 2010.

[28] J. Tesson and F. Loulergue, "A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation," in *International Conference on Computational Science (ICCS)*. Singapore: Elsevier, 2011, pp. 36–45.

[29] L. Gesbert, Z. Hu, F. Loulergue, K. Matsuzaki, and J. Tesson, "Systematic Development of Correct Bulk Synchronous Parallel Programs," in *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 2010, pp. 334–340.

[30] F. Loulergue, S. Robillard, J. Tesson, J. Légaux, and Z. Hu, "Formal Derivation and Extraction of a Parallel Program for the All Nearest Smaller Values Problem." Gyeongju, Korea: ACM, 2014, pp. 1577–1584.

[31] K. Emoto, F. Loulergue, and J. Tesson, "A Verified Generate-Test-Aggregate Coq Library for Parallel Programs Extraction," in *Interactive Theorem Proving (ITP)*, ser. LNCS, no. 8558. Wien, Austria: Springer, 2014, pp. 258–274.

[32] F. Loulergue, "A verified accumulate algorithmic skeleton," in *Fifth International Symposium on Computing and Networking (CANDAR)*. Aomori, Japan: IEEE, November 19-22 2017, pp. 420–426.

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

[34] F. Gava, "A modular implementation of data structures in bulk-synchronous parallel ML," *Parallel Processing Letters*, vol. 18, no. 1, pp. 39–53, 2008.

[35] F. Loulergue, "Implementing Algorithmic Skeletons with Bulk Synchronous Parallel ML," in *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 2017, to appear.

[36] F. Loulergue, V. Niculescu, and J. Tesson, "Implementing powerlists with Bulk Synchronous Parallel ML," in *Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. Timisoara, Romania: IEEE, 2014, pp. 325–332.

[37] V. Allombert, F. Gava, and J. Tesson, "A formal semantics of the Multi-ML language," in *International Symposium on Parallel and Distributed Computing*. IEEE, Jun. 2018, to appear.

[38] V. Allombert and F. Gava, "An ML implementation of the MULTI-BSP model," in *International Conference on High Performance Computing and Simulation*, 2018, to appear.

[39] Bourgoin, Mathias and Chailloux, Emmanuel and Lamotte, Jean-Luc, *Experiments with Spoc*. Saxe-Coburg Publications, 2015.

[40] Bourgoin, Mathias and Chailloux, Emmanuel, "GPGPU composition with OCaml," in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 2014, p. 32.

[41] Bourgoin, Mathias and Chailloux, Emmanuel and Lamotte, Jean-Luc, "High Level Data Structures for GPGPU Programming in a Statically Typed Language," *International Journal of Parallel Programming*, vol. 45, no. 2, pp. 242–261, 2017.

[42] Bourgoin, Mathias and Chailloux, Emmanuel and Doumoulakis, Anastasios, "Profiling High Level Heterogeneous Programs," in *Tenth International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG 2017)*, 2017.

[43] Bourgoin, Mathias and Chailloux, Emmanuel, "High-level accelerated array programming in the web browser," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 2015, pp. 31–36.