An ML implementation of the MULTI-BSP model

Victor Allombert, Frédéric Gava

An ML implementation of the MULTI-BSP model

Victor Allombert
Université d'Orléans, LIFO
Orléans, France
victor.allombert@univ-orleans.fr

Frédéric Gava
Université Paris-Est Créteil, LACL
Créteil, France
gava@u-pec.fr

***Abstract*—We have designed a parallel language called MULTI-ML for programming MULTI-BSP algorithms in ML. The MULTI-BSP model provides a tree-based view of nested components of hierarchical architectures. The structure and abstraction brought by MULTI-BSP allows for portable parallel programs with scalable performance predictions, without dealing with low-level details of architecture. The MULTI-ML language is essentially based on the concept of multi-functions which are recursive functions used to execute perform recursions on the nested sub-components through the MULTI-BSP (hierarchical) architecture. In this paper we design a generic compilation scheme of the MULTI-ML language dedicated to parallel machines performing data exchanges using synchronous requests, derived from any ML compilation. This ensures that the implementation follows the semantics, allowing greater confidence in the safety of execution.**

***Keywords*—Hierarchical architectures; MULTI-BSP; compilation; ML**

I. Introduction

To be compliant with a *bridging model* simplifies the way of writing code and ensures *efficiency* and *portability* from one architecture to another.

Our previous work aimed at designing a parallel functional language based on the BSP (Bulk Synchronous Parallel) bridging model [1]: BSML [2].

A BSP machine is a set of processors and memory pairs communicating through a common network. The machine is thus mapped on a parallel architecture and propose a structured way to program it. A BSP program consists in the execution of a sequence of super-steps. As described in Fig. 1, a super-steps is composed by three phases: a local computation; a communication phase concerning all the units of the BSP machine; and finally, a global synchronisation of all the computing units. After this synchronisation, the communicated values are available and the next super-steps can start. Thanks to this quasi-synchronous execution scheme, deadlocks are avoided and determinism is preserved. The model also ensures efficiency and portability from one architecture to another.

As BSML is a library for the OCAML language, it relies on a powerful static type system. Thus, the code generated by the compiler is safe, thanks to the type checker. A byte-code compiler is available to allow code portability on various architectures and performances close to the highest standards of modern compilers. Thanks to the functional approach brought by BSML, it is possible to extract a certified BSML parallel program from the specification defined with the COQ proof assistant [3].

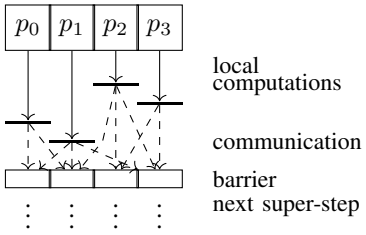

Fig. 1. A BSP super-step.

However modern parallel architectures are hierarchical and have multiple layers of parallelism: super-computers are made by thousands of interconnected nodes, each one carrying several multi-core processors. Communication between distant nodes may not be as fast as communication among the cores of a given processor. Indeed, the communication between cores is done by accessing the shared cache of cores, which is much faster than communication between processors through RAM. As BSP was designed for *flat* architectures, we now consider the MULTI-BSP model [4], an extension of BSP which is dedicated to *hierarchical* architectures with multiple levels of memories (or networks).

MULTI-BSP aims to take advantage of hierarchical architecture in order to achieve better performances and preserve the BSP properties. The MULTI-BSP model introduced a representation where a hierarchical architecture is a tree structure of nested components (sub-machines) where the lowest stage (leaves) are processors and every other stage (nodes) only contains memory. Figure 2 shows the hierarchical point of views brought by MULTI-BSP compared to BSP for a multi-core made of 2 cores with 4 threads per core.

MULTI-ML [5], [6] is an extension of ML (OCAML [7]) for programming MULTI-BSP algorithms using a small set of primitives (as BSML was, but limited to BSP algorithms).

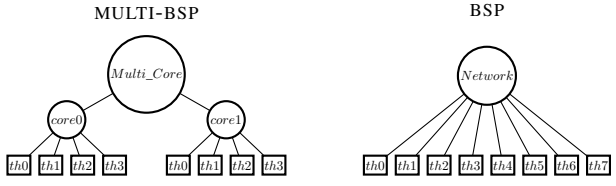

Fig. 2. The difference between the MULTI-BSP and BSP models for a multi-core architecture.

Our current MULTI-ML implementation relies on MPI communication routines. There is no study showing that this implementation is faithful to the MULTI-ML's semantics proposed in [8]. Furthermore, having only an MPI implementation is not sufficient. We can imagine a particular machine where a specific library can take advantage of the architecture to optimise the communications. In such a context, it is more suitable to use a correct and generic compilation scheme for any machine with specific communication primitives to benefit from the low level optimisations of any standard (sequential) ML compiler. Such a compilation scheme could abstract the low level primitives needed to implement the language features thus, allowing a portable implementation — with potential optimisations. This ensures a greater confidence for the implementation of the compiler and thus, for the safety of the language.

We first briefly present our past implementation and give an informal presentation of MULTI-ML in the form of a core language. For sake of conciseness, we do not detail the MULTI-BSP model nor MULTI-ML language; They are described in [6], [5] and the current implementation can be found at https://git.lacl.fr/vallombert/Multi-ML. The main idea of MULTI-ML is to run a BSML-like code (a BSP code in ML) on each level of the MULTI-BSP (hierarchical) tree of components. Then we define the new compilation scheme. Finally, we give some correctness results and conclusion.

II. The previous implementation

A. Several daemons on processors

The past implementation of [6] works in a SPMD (Single Program Multiple Data) fashion and the compiler generates OCAML [7]+MPI [9] code from a source program written in MULTI-ML. There is one MPI process for each node and leaf of the MULTI-BSP architecture. Each process can be seen as a "daemon" which is waiting for execution *orders*. Basically, an *order* consists of a piece of code which must be evaluated by the process concerned. The communications between daemons are done using asynchronous send and receive routines. The communicated values are composed by serialised closures which contain a code and a set of values needed for their evaluation. Thus, a serialised closure is a self-contained value that can be evaluated after its communication, in an other evaluation context. Any daemon un-serialises the closures and executes them with the appropriate arguments. For example, the following code `<<#x#+1>>` is compiled into the closure $(\mathbf{fun}\ \ x \rightarrow x+1)$. Both the closure and the value of `x` are sent to the nested nodes of the MULTI-BSP components.

This *"naive"* implementation has the advantage of being quite simple to implement and relatively efficient in most cases. It does however have several drawbacks: (1) The closure to be sent can be large if it contains many ML codes; in an SPMD model, one might want only the necessary values (in this case, the value of `x`); (2) In case of a MULTI-BSP tree with many levels, every node above the leaves is a process. In addition to an unnecessary scheduling, these processes are also executed by the computing units (the cores) as MPI processes; (3) For nested processes executed on the same unit, an unnecessary transmission is performed, as values are already available, since they share the same computing unit.

The goals of the new implementation are: (1) Minimise the number of daemons by having only one process on each computing unit; We use continuations (threading) in place of multiple MPI processes; (2) Abstract the communications (like in BSML [2]) in order to get only one routine module [2] to implement using any communication library; (3) Abstract the sequential part of the compiler (OCAML). To do so, we will extend the traditional compilation scheme of ML with MULTI-ML routines and then, we will describe the compilation scheme of MULTI-ML codes on any abstract machine.

A subtlety in the cost analysis of a program is the binding of global values, that is, values evaluated outside multi-functions and seemingly accessible everywhere. But the programmer could encounter unexpected behaviours which can *break* the cost model. Take for example the following code:

```
let v = big_value;; (* partial end of evaluation *)

let multi f () =
 where node =
  ... << v >> ... (* a BSML code *)
 where leaf = ... (* an OCaml code *)
  ...
```

We assume here that `v` is a "large value" (that we clearly do not want to transmit). Then, inside the scope of a parallel vector (that is to say within the `<< >>` notation), referencing `v` does not imply additional communication, as `v` is defined in the MULTI-BSP (global) memory. The serialisation procedure simply gives the address of the value `v`, as the code is SPMD. In this case, there is no unexpected behaviour. But now take this relatively close code:

```
let g () =
 let v = big_value in
 let multi f _ =
  where node =
   ... << v >> ...
  where leaf =
  ...
```

On the contrary, the value `v` is evaluated when the function `g` is applied. Therefore, the address of the value `v` is not known by the processes and can differ: the complex serialisation mechanism of OCAML forces the value to be transmitted inside the function closure, when referenced inside the scope of a parallel vector. In this particular case, an unexpected communication occurs: the value `v` is completely communicated. This additional communication is hidden from the programmer which contradicts the MULTI-BSP's explicit cost model.

Thus, if the values are not declared as a *list of expressions*, it is not possible to reference them within the scope of a parallel vector. The value must be completely evaluated; otherwise, it is not possible to determine which identifier (or address) corresponds to a particular value. Indeed, in the case of nested bindings, it is not possible to determine a *global* identifier to reference a value. It is thus impossible to tell a process which value to use. To avoid this behaviour, our compilation scheme will prohibit the usage of such variables in the scope of a parallel vector.

$$
\begin{array}{rl}
e ::= & \textit{Expressions with core-ML} \\
& \mid x \mid \mathbf{cst} \mid \mathbf{op} \mid (e,e) \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid (e\ e) \\
& \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid (\mathbf{fun}\ x \rightarrow e) \mid (\mathbf{rec}\ f\ x \rightarrow e) \\
& \textit{and BSML-like primitives} \\
& \mid \mathbf{mkpar}\ e \mid \mathbf{gid} \mid \mathbf{proj}\ e \mid \mathbf{put}\ e \mid \mathbf{pid} \mid (\mathbf{copy}\ x) \\
& \mid \mathbf{replicate}\ (\mathbf{fun}\ \ f \rightarrow e) \mid \mathbf{apply}\ e\ e \\
& \textit{and multi-functions} \\
& \mid (\mathbf{multi}\ \ f\ x \rightarrow e_1 \dagger e_2) \\
P ::= & \textit{Programs} \\
& \mathbf{let}\ x = e \mid \mathbf{let}\ x = e\ ;;\ P
\end{array}
$$

Fig. 3. Syntax of core-MULTI-ML.

B. A core language for MULTI-ML

Fig. 3 gives the syntax of the core-MULTI-ML which extends the popular core-ML. The core-MULTI-ML syntax stands for the minimal set of syntactical constructions needed to write a MULTI-ML program. A program is a list of expressions which are executed one after the other. Expressions contain variables, constants (integers, *etc.*), operators ($\leq$, $+$, *etc.*), pairs, the standard ML statements **let**, **if**, **fun**, **rec** for recursive function calls, the BSML-like primitives [5], [6], [2] (**replicate**, **put**, **proj**, **mkpar**, **apply**), local copy of a parent's variable to its children (**copy**) and a tree of identifiers (**gid**). We define "let-multi" as a particular function construction $\mathbf{multi}\ \ f\ x \rightarrow e_1 \dagger e_2$ with codes for both the nodes (e_1) and the leaves (e_2). Only the expressions without free variables in $\mathbf{replicate}\ (\mathbf{fun}\ g \rightarrow e)$ are valid as explained above. The set of free variables for an expression e is as usual and is denoted $\mathcal{F}(e)$.

`replicate` is well defined when the g in $\mathbf{replicate}\ (\mathbf{fun}\ g \rightarrow e)$ is in the scope of a multi-function only. We note $\mathcal{WF}\_(e)$ a well formed expression inductively defined in Fig. 4. It ensures that a multi-function can be nested in the sub-components of the hierarchical machine only. For example, it forbids codes like $\mathbf{fun}\ f \rightarrow \mathbf{replicate}\ (\mathbf{fun}\ f \rightarrow e)$ because the sub-component (which could be a distant machine) cannot know the function (multi or not) when evaluated `e` without this function being serialised which can lead to unnecessary communications.

We do not use the syntactic sugars [2] of BSML/MULTI-ML and assume that they have been transformed into primitive calls first: (1) $\ll e \gg \equiv \mathbf{replicate}(\mathbf{fun}\ x \rightarrow e)$; (2) every access $\$x\$$ (inside a vector) to the local value of a vector x has been replaced by a call of **apply**, *e.g.* $\ll \$x\$ + 1 \gg \equiv \mathbf{apply}\ x\ (\mathbf{replicate}\ (\mathbf{fun}\ \_\ \ x\ \ \rightarrow\ \ x + 1))$, where `_` matches all possible values; (3) in the same way, each local copy $\#x\#$ of a parent's variable x is replaced by a call to the `copy` primitive. The **replicate** and **apply** primitives are available in the core syntax only. For the sake of programming simplicity, the syntactic sugars are available to the programmers only.

$$
\begin{array}{rcl}
\mathcal{WF}_f(x) & = & \mathbf{true} \\
\mathcal{WF}_f(\mathbf{cst}) & = & \mathbf{true} \\
\mathcal{WF}_f(\mathbf{op}) & = & \mathbf{true} \\
\mathcal{WF}_f((e_1,e_2)) & = & \mathcal{WF}_f(e_1) \wedge \mathcal{WF}_f(e_2) \\
\mathcal{WF}_f(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) & = & \mathcal{WF}_f(e_1) \wedge (\mathcal{WF}\_(e_2)) \\
 & & \text{if } (x \equiv f) \\
\mathcal{WF}_f(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) & = & \mathcal{WF}_f(e_1) \wedge (\mathcal{WF}_f(e_2)) \\
 & & \text{otherwise} \\
\mathcal{WF}_f(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) & = & \mathcal{WF}_f(e_1) \wedge \mathcal{WF}_f(e_2) \\
 & & \wedge \mathcal{WF}_f(e_3) \\
\mathcal{WF}_f((\mathbf{fun}\ x \rightarrow e)) & = & \mathcal{WF}\_(e) \quad \text{if } (x \equiv f) \\
\mathcal{WF}_f((\mathbf{fun}\ x \rightarrow e)) & = & \mathcal{WF}_f(e) \quad \text{otherwise} \\
\mathcal{WF}_f((\mathbf{rec}\ f\ x \rightarrow e)) & = & \mathcal{WF}\_(e) \quad \text{if } (x \equiv f \vee g \equiv f) \\
\mathcal{WF}_f((\mathbf{rec}\ f\ x \rightarrow e)) & = & \mathcal{WF}_f(e) \quad \text{otherwise} \\
 & & \textit{BSML-like primitives} \\
\mathcal{WF}_f(\mathbf{replicate}\ (\mathbf{fun}\ g \rightarrow e)) & = & \mathbf{true} \quad \text{if } (g \equiv f) \\
\mathcal{WF}_f(\mathbf{replicate}\ (\mathbf{fun}\ g \rightarrow e)) & = & \mathbf{false}\ \text{otherwise} \\
\mathcal{WF}_f(\mathbf{copy}\ x) & = & \mathbf{true} \\
\mathcal{WF}_f(\mathbf{mkpar}\ e) & = & \mathcal{WF}_f(e) \\
\mathcal{WF}_f(\mathbf{put}\ e) & = & \mathcal{WF}_f(e) \\
\mathcal{WF}_f(\mathbf{proj}\ e) & = & \mathcal{WF}_f(e) \\
\mathcal{WF}_f(\mathbf{apply}\ e_1\ e_2) & = & \mathcal{WF}_f(e_1) \wedge \mathcal{WF}_f(e_2) \\
 & & \textit{multi-functions} \\
\mathcal{WF}\_((\mathbf{multi}\ f\ x \rightarrow e_1 \dagger e_2)) & = & \mathcal{WF}_f(e_1) \wedge \mathcal{WF}\_(e_2)
\end{array}
$$

Fig. 4. Well formed terms of core-MULTI-ML.

III. A GENERIC COMPILATION SCHEME FOR MULTI-ML

A. A new compilation scheme

To get a generic compilation of MULTI-ML, we must not depend on the compilation of the sequential parts of the language. Our goal is to generate pure OCAML codes with some low level communication routines. We abstract such a compilation with the notation $[\![e]\!]_{\text{s}}$, corresponding to the compilation of the term e with any standard ML compiler. In general, such a compilation generates instructions manipulating a stack and some environments (memories). We denote them $\mathcal{E}$ with the **pid** (or processor identifier) of each core of the machine as subscript (assuming an unique **pid** for each core of the machine). We note $[\![e]\!]_{\text{M}}^{-}$ the compilation of a MULTI-ML expression. This compilation may have called $[\![e]\!]_{\text{s}}$, if a sub-term is free of MULTI-ML primitives. Without considering the type system of MULTI-ML, it is hard to detect such expressions statically. In our case, and without lack of generality, we prefer to apply such a compilation to e after the pass of $[\![e]\!]_{\text{M}}^{-}$[1].

The compilation of a term e is thus a compilation which transforms all MULTI-ML features into standard ML ones and then, compile them using the standard ML compilation. The compilation of a program $P \equiv e_1;;\cdots;;e_n$ is the compilation of all the expressions, that is, $[\![P]\!]_{\text{M}} = [\![e_1]\!]_{\text{M}}^{-};;\cdots;;[\![e_n]\!]_{\text{M}}^{-}$. The compilation can be expressed as following: $\langle\!\langle \{\mathcal{E}_0, P'\}, \ldots, \{\mathcal{E}_{\mathbf{p_c}}, P'\} \rangle\!\rangle$ where $\mathbf{p_c}$ stands for the total number of cores minus one of the MULTI-BSP machine and where $P' = [\![[\![P]\!]_{\text{M}}]\!]_{\text{s}}$.

The execution scheme of a program P is based on the following ideas: (1) The code is duplicated to be executed on each core; the hierarchical architecture is thus *"flattened"*

[1] We left the possibility of optimisations for future work.

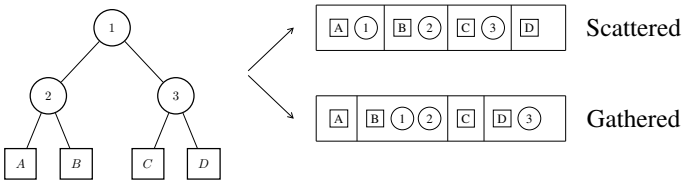

Fig. 5. Architecture *flattening*.

with the hypothesis that on the same core, only nodes and leaves of the same branch may appear. Moreover, each node is managed by one leaf. The Fig. 5 represents a three level architecture – here, we have a multi-core (1) with two cores $(2, 3)$ and two threads per cores (A, B, C and D) – where we describe two different ways of mapping the daemon processes to physical cores: the *scattered* way aims to balance the processes over the physical cores; the *gathered* way aims to gather the processes on a minimal set of physical cores. In the current implementation of MULTI-ML the distribution is made statically using the *scattered* flattening. (2) The code outside multi-functions is executed sequentially, once, for each core; (3) A multi-function is a special function which initialises some data (basically, it is defined by an identifier corresponding to the codes of nodes and leaves, stored in a global hash table of multi-functions) and runs a scheduler with the appropriate code of the root node; (4) Each scheduler is waiting for two kinds of instructions: spawning a daemon and continuing the execution or terminating a multi-function execution; Daemons are thus independent, which allows for different daemons running on the same physical core; each daemon corresponding to a node or a leaf during both the *"upward"* and *"downward"* phase of the recursive call of the multi-function; (5) Each daemon is waiting for instructions from its *parent* (upper-node), which are: (a) Perform an asynchronous BSML-like routine such as a **`replicate`** or **`apply`**; (b) Perform a synchronous routine such as **`proj`**, **`mkpar`**, **`copy`** or **`put`**; (c) Complete the computation, since the parent has also completed the node code. (6) Leaf codes are running sequentially; (7) A multi-function recursive call is *"just"* the execution of the code corresponding to the identifier of a multi-function. The main difficulties come from parallel vectors, their primitives and multi-functions.

B. Identifiers for parallel vectors and multi-functions

From the node point of view, a parallel vector contains a code which manages the memories of distant cores. To avoid the communication of serialised codes (which are known and shared by each core), the construction of a parallel vector is based on the following idea: we give a *static identifier* for each parallel vector in the program (syntactically, the code following **`replicate`**). When created, a parallel vector uses a global hash table that relates those identifiers to the corresponding codes. Then a *dynamic identifier* is shared between a node (parent) and its sub-nodes (children), allowing to reference a parallel vector between the two processes. It is thus possible to remove a parallel vector when it is detected as obsolete by the parent's node (using a garbage collection procedure).

A multi-function is not a simple function since it can be propagated through the whole hierarchical machine. A simple closure is thus not satisfying. The system must keep trace of which multi-function is currently running, using a global variable called **`CurrentIdM`**. We write **`CodeId`** for the identifier of each core of the machine. A multi-function is compiled as follow:

```
(*Multi function "where node =e1 where leaf =e2"*)
(fun mf x →
let idM= newIdM() in
let f = (fun i x → CurrentIdM←idM;
  if node(i) then begin
    WakeUpChildren|i| CoreId;
    let v=e1 in Signal|i| EndNode; v
  end
  else e2) in
|HM|[idM]←f;
if CoreId=root then run (WakeUpAll (f |0̃| x));
let v = scheduler() in |HM|[idM]←null; v)
```

Where $|i|$ is standing for the component identifier and where $|\tilde{0}|$ stands for the root node. Notice that, due to the execution model of MULTI-ML, all cores are running the exact same code and thus, all of them are registering the same multi-functions in order to execute them (in a SPMD way). To do so, an identifier is first created (**`newIdM`**). Then, we create a specific function with an argument `i` that will be the **`gid`** (global identifier) of the current level of execution. The **`CurrentIdM`** is then modified by the identifier of the multi-function called. If the process is a node, the node code of the multi-function is executed, after notifying the sub-nodes (using **`WakeUpChildren`**) to wait for instructions (**`Signal`**). Then daemons will be used to manipulate the vectors (construction, projection, communication, *etc.*). Otherwise, when the process is a leaf, the leaf code is executed. This function is registered in the global table of multi-functions $\mathcal{HM}$. When the multi-function starts, the root node is active and all other processes are awakened (**`WakeUpAll`**), waiting for orders. Finally, the multi-function is un-registered (globally by all the cores) and we return the computed value. The code of the scheduler is:

```
let scheduler () = match wake() with
  | Dmn i → run daemon|i|(); scheduler();
  | EndMulti v → v
```

Here, **`wake`**`()` is a *passive-wait* function which waits until an order is received. Note that each core has its own ordered queue of messages. That could be easily implemented using a library such as MPI.

C. Primitives and their interaction with daemons

Now, we have to handle parallel vector manipulations (BSML-like primitives). The content of the vectors is handled by the children, using a hash table of vectors where the identifiers are the keys. From the parent point of view, a parallel vector is just a dynamic identifier. For sake of conciseness, we give only the code of the **`replicate`** and **`proj`** primitives in Fig. 6, other codes from the rest of the primitives can be easily deduced and are fully available in [5].

```
let replicate = fun idS →
 let idD= newIdD() inj(idD);
 let idM = CurrentIdM in
  Signal|^i| Rpl(idD,idS,idM);
 idD
```

```
let proj = fun idD →
 Signal|^i| Prj(idD);
 let tab=
  fromChildren|^i|() in
  (fun j → tab[j])
```

Fig. 6. Compilation of BSML-like primitives.

For **replicate**, the parameter is the static identifier used to build the parallel vector. We will explain, later, how we obtain it. Here, two identifiers are necessary: (1) the dynamic identifier of the current vector and (2) the identifier of the current (running) multi-function.When a parallel vector construction order is received (`Rpl`), three identifiers must be communicated to the children that are, by construction, daemons which are awaiting orders. The identifier of the multi-function is used to bind the call of the multi-function in the body of replicate. As the **replicate** routine is asynchronous, the **Signal** routine is an asynchronous send which involves insignificant costs for modern architectures and many cores. Finally the generated dynamic identifier is returned to complete the **replicate**.

The code for the **proj** is similar, except for the use of a synchronous routine to communicate the values of the children upwards: the routine **fromChildren** generates an array containing the values received from each process. The order "Prj" forces the children to communicate their values. The resulting value is thus, for each i, a function mapping of those values.

Finally, the code of a daemon consists in waiting for orders, executing them and waiting until the execution ends.

```
let rec daemon|^i|() = match rcv|^i| with
| Rpl(idD,idS,idM) →
  |VD|[idD]←(|VS|[idS] |VM|[idM]);
  daemon|^i|()
| Prj(idD) → up|^i| |VD|[idD]; daemon|^i|()
| ... (* other BSML-like primitives *)
| EndNode → ()
```

In this code, we have: (1) $\mathcal{VD}$, $\mathcal{VM}$ and $\mathcal{VS}$ are, respectively, the hash table of the dynamic identifier of parallel vectors, of multi-function identifiers and the shared hash table of static identifiers (from identifiers to codes); (2) "up" which sends the value to the parent and thus synchronises the children.

The code works as follows: When the `Rpl` order (for **replicate**) is received, a new parallel vector is created with the given information. Regarding the `Prj` (for **proj**) order, the value of the vector is communicated upwards by reading the hash table. The other BSML-like primitives works similarly.

Note that the use of only one daemon for simulating the execution of upper nodes seems impossible. Indeed, if a process must manage two nodes, the daemon simulating the first node may be waiting for the results of its children, which could be itself. Moreover the second node could also wait for the results of its own children, resulting in a situation of deadlock if no more than one thread is active. To avoid such problems, we choose a solution where by on each core, only one thread is active during all the execution of the MULTI-ML code.

Sequential computations

$$
\begin{array}{rcl}
[\![x]\!]^g_{\text{M}} & = & x \quad \text{if } x \neq g \\
[\![x]\!]^g_{\text{M}} & = & (\mathbf{call}\ g) \\
[\![\mathbf{cst}]\!]^g_{\text{M}} & = & \mathbf{cst} \\
[\![\mathbf{op}]\!]^g_{\text{M}} & = & \mathbf{op} \\
[\![(e_1, e_2)]\!]^g_{\text{M}} & = & ([\![e_2]\!]^g_{\text{M}}, [\![e_1]\!]^g_{\text{M}}) \\
[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]^g_{\text{M}} & = & \mathbf{let}\ x = [\![e_1]\!]^g_{\text{M}}\ \mathbf{in}\ [\![e_2]\!]^g_{\text{M}} \quad \text{if } x \neq g \\
[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]^g_{\text{M}} & = & \mathbf{let}\ x = [\![e_1]\!]^g_{\text{M}}\ \mathbf{in}\ [\![e_2]\!]^-_{\text{M}} \quad \text{otherwise} \\
[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]^g_{\text{M}} & = & \mathbf{if}\ [\![e_1]\!]^g_{\text{M}}\ \mathbf{then}\ [\![e_2]\!]^g_{\text{M}}\ \mathbf{else}\ [\![e_3]\!]^g_{\text{M}} \\
[\![(\mathbf{fun}\ x \to e)]\!]^g_{\text{M}} & = & (\mathbf{fun}\ x \to [\![e]\!]^g_{\text{M}}) \quad \text{if } x \neq g \\
[\![(\mathbf{fun}\ x \to e)]\!]^g_{\text{M}} & = & (\mathbf{fun}\ x \to [\![e]\!]^-_{\text{M}}) \quad \text{otherwise}
\end{array}
$$

BSML-*like primitives*

$$
[\![\mathbf{replicate}\ (\mathbf{fun}\ g \to e)]\!]^g_{\text{M}} = \text{Code given above with } idS, \mathcal{H} \oplus \{idS \Rightarrow (\mathbf{fun}\ g \to [\![e]\!]^g_{\text{M}})\} \text{ where } idS \text{ is a fresh value}
$$

multi-functions

$$
[\![(\mathbf{multi}\ f\ x \to e_1 \dagger e_2)]\!]^-_{\text{M}} = \text{Code given above where } e'1 = [\![e_1]\!]^f_{\text{M}} \text{ and } e'2 = [\![e_2]\!]^f_{\text{M}}
$$

Fig. 7. Compilation scheme of core-MULTI-ML.

Fig. 7 defines the compilation $[\![e]\!]^-_{\text{M}}$ of an expression e. It works as follows:

- Most sequential constructions are left as they are or are built over trivial inductions. This is the case for constants, operators, pairs, conditional, bindings and functions. For the BSML-like primitives, the compilation corresponds to the code given above;
- At the beginning, we are outside the scope of a multi-function binding. But after the declaration of a multi-function, we must select the appropriate identifier which is given to the compilation function; When executing the **replicate** primitive, g stands for the current evaluated multi-function and thus, it is used to define where the call to the **CurrentIdM** must be made. Moreover, we update the hash table $\mathcal{H}$ of vectors.

Thus, the compilation of a program begins with $\mathcal{H} \equiv \emptyset$. Note that our core-language does not allow the definition of multiple multi-function at the same time. To overcome this limitation, an appropriate set of bindings must be given as arguments of **replicate**, corresponding to all the potential multi-function calls. This feature is discussed in future works.

D. Execution and Correctness

The execution of a compiled program is done using a small-steps semantics. It consists of a predicate between two expressions defined by a set of rules called *steps*. As usual, we have two kinds of reductions. One for the cores and one for the whole machine. We note $\Rightarrow_c$ the local reduction of an expression on a single core and $\Rightarrow$ the reduction of a whole machine expression. For both rules, $\Rightarrow^*$ is a reflexive and transitive closure and $\Rightarrow^\infty$ is used for infinite programs. The reduction of a program $P = e_1;;\cdots;;e_n$ consists of the reduction from e_1 to e_n.

For sake of simplicity, we do not give the reduction rules of OCAML expressions. We focus on the reduction of communication routines needed for the implementation of MULTI-ML. Note that sequential codes neither modify the communication environments nor the message queues. An environment $\mathcal{E}$

$$
\begin{array}{lcll}
\Gamma^i & ::= & [] & \textit{head evaluation} \\
 & | & \Gamma^i e & \textit{application} \\
 & | & v\Gamma^i & \textit{application} \\
 & | & \textbf{let } x = \Gamma^i \textbf{ in } e & \textit{let} \\
 & | & (\Gamma^i, e) & \textit{left pair} \\
 & | & (v, \Gamma^i) & \textit{right pair} \\
 & | & \textbf{if } \Gamma^i \textbf{ then } e \textbf{ else } e & \textit{conditional}
\end{array}
$$

Fig. 8. Abstract context syntax.

contains at least two independent queues of orders, one for the schedulers (denoted $\mathcal{F}^s$) and another shared by the daemons (denoted $\mathcal{F}^d$). We write $\lhd e \rhd$ for a thread that runs the expression e. We note $|^v_\vdots|$ for reading the message v at the top of a queue and $|^\vdots_v|$ for adding a message v at the bottom of a queue.

It is easy to see that we cannot always perform a head reduction of an expression. We have to reduce, in depth, the sub-expression. To define such a depth reduction, we define two kinds of contexts: (1) an expression Γ^i with a "hole" denoted [] which have the common syntax for ML expressions (see [5] for more details) where i is the **gid** of the expression; (2) we then define Γ^i_j (given in Fig. 8) as the context of each threads where the reduction holds. For the core j at gid i, $\{\mathcal{E}_j, \Gamma^i_j[e]\}$ stands for $\{\mathcal{E}_j, \lhd e' \rhd, \ldots, \lhd \Gamma^i[e] \rhd, \ldots \lhd e'' \rhd\}$. From the previously defined low level routines, we have the following rules:

a) Signal and rcv.: The **Signal** function is a low level asynchronous communication routine which aims to send a message to the queue $\mathcal{F}^d$ of another (distant or not) thread. Then the **rcv** routine can read it. We write $\mathcal{E} \oplus |^\vdots_\vdots|^i$ to highlight the use of the queue of daemons (allocated to **gid** i) without modifying the rest of the environment. First we define the sending of a message:

$$\langle\!\langle \ldots, \{\mathcal{E}_0 \oplus |^\vdots_\vdots|^i, e^i_0\}, \ldots, \{\mathcal{E}_{\mathbf{p}} \oplus |^\vdots_\vdots|^i, e^i_{\mathbf{p}}\}, \{\mathcal{E}, \Gamma^i_j[\mathbf{Signal^i}\ v]\}, \ldots \rangle\!\rangle$$

$$\Rightarrow_c \langle\!\langle \ldots, \{\mathcal{E}_0 \oplus |^v_\vdots|^i, e^i_0\}, \ldots, \{\mathcal{E}_{\mathbf{p}} \oplus |^v_\vdots|^i, e^i_{\mathbf{p}}\}, \{\mathcal{E}, \Gamma^i_j[()]\}, \ldots \rangle\!\rangle$$

Where v is one of the possible messages presented above and where $\{0, \ldots, \mathbf{p}\}$ are the global identifiers of the siblings of core j. Note that the message might be processed later. For example, such a case may occur when a daemon receives different messages manipulating a parallel vector:

```
let v1=replicate (fun g → 1) in
let v2=replicate (fun g → 2) in
let vplus=replicate (fun g → (+) ) in
 (apply (apply vplus v1) v2)
```

As daemons work asynchronously, one of them may have received all the messages before executing the orders. Secondly, we define the rule of reception:

$$\langle\!\langle \ldots, \{\mathcal{E}_j \oplus |^\vdots_v|^i \lhd \Gamma^{i'}_j \rhd \cdots \lhd \Gamma^i_j[\mathbf{rcv^i}()] \rhd \cdots, \ldots \rangle\!\rangle \Rightarrow_c$$

$$\langle\!\langle \ldots, \{\mathcal{E}_j \oplus |^\vdots_\vdots|^i \lhd \Gamma^{i'}_j \rhd \cdots \lhd \Gamma^i_j[v] \rhd \cdots, \ldots \rangle\!\rangle$$

On core j, the daemon identified by i reads a message in its own queue. Note that these rules are not global since they only rely on the thread of a single core, instead of synchronous rules that require the whole machine to work.

b) up.: To communicate a value upward (from children to a parent) in a synchronous way, we have the following rule:

$$\langle\!\langle \ldots, \{\mathcal{E}_0, \Gamma^i_0[\mathbf{up}\ v_0]\}, \ldots,$$

$$\{\mathcal{E}_{\mathbf{p}}, \Gamma^i_{\mathbf{p}}[\mathbf{up}\ v_{\mathbf{p}}]\}, \{\mathcal{E}, \Gamma'^i_j[\mathbf{fromChildren}()]\}, \ldots \rangle\!\rangle \Rightarrow$$

$$\langle\!\langle \ldots, \{\mathcal{E}_0, \Gamma^i_0[()]\}, \ldots, \{\mathcal{E}_{\mathbf{p}}, \Gamma^i_{\mathbf{p}}[()]\}, \{\mathcal{E}, \Gamma'^i_j[[|v_0, \ldots, v_{\mathbf{p}}|]]\}, \ldots \rangle\!\rangle$$

where $\{0, \ldots, \mathbf{p}\}$ are the global identifiers of the siblings of core j.

c) run: is a routine used to run threads (daemons):

$$\langle\!\langle \ldots, \lhd \Gamma^{i'}_j \rhd \cdots \lhd \Gamma^i_j[\mathbf{run}\ d^k] \rhd \cdots, \ldots \rangle\!\rangle$$

$$\Rightarrow_c \langle\!\langle \ldots, \lhd \Gamma^{i'}_j \rhd \cdots \lhd \Gamma^i_j[()] \rhd \lhd d^k \rhd \cdots, \ldots \rangle\!\rangle$$

Where, on core j, for **gid** i, we run the daemon for **gid** k. When a thread ends, it may have produced an *orphan* value. When such a value is not communicated upward using the **proj** primitive, there is no need to keep them. Thus:

$$\langle\!\langle \ldots, \lhd e \rhd \cdots \lhd v \rhd \cdots, \ldots \rangle\!\rangle \Rightarrow \langle\!\langle \ldots, \lhd e \rhd \cdots, \ldots \rangle\!\rangle$$

d) WakeUpChildren and WakeUpAll: They are working in the same way. **WakeUpAll** aims to send messages to all cores of the architecture, whereas **WakeUpChildren** sends messages to its children only. To do so, these routines send messages to the queue $\mathcal{F}^s$ of each involved thread. For the sake of conciseness, we do not detail the rule.

There are also rules for both upward and downward communication between parent and children. They are given in [5] and can be deduced easily from the rules above.

e) From local to global reductions.: Finally, the asynchronous reduction $\Rightarrow_c$ (for standard OCAML expressions) can only work in the context of a call of the $\Rightarrow$ reduction on the whole machine, using the following rule:

$$\langle\!\langle \ldots, \{\mathcal{E}_j, \cdots \lhd \Gamma^i_j[e] \rhd \cdots\}, \ldots, \rangle\!\rangle$$

$$\Rightarrow \langle\!\langle \ldots, \{\mathcal{E}_j, \cdots \lhd \Gamma^i_j[e'] \rhd \cdots\}, \ldots, \rangle\!\rangle$$

If $\Gamma[e] \Rightarrow_c \Gamma[e']$ uses a standard reduction rule of OCAML (matching, if-then-else, application, binding, pairing, *etc.*). Note that all the low level communication routines only works when sending valid values, as in the operational semantics. We do not introduce a validity test for values to simplify the formalisation.

Algorithm	i7	mirev3_1	mirev3	mirev3_{ht}
TDS	690	2070	8625	16100
FFT	972	2916	12150	22680

TABLE I
FUNCTIONS CLOSURES OVERHEAD (IN BYTES)

E. Results

1) Correctness of the compilation scheme: Now we have the following results that can be proved by induction and co-induction.

Theorem 1. *For a program* $P = e_1;;\cdots;;e_n$ *and if* $\mathcal{WF}_{\_}(e_i)$,

- *If* $\mathcal{M} \vdash e_i \Downarrow_p^{\mathcal{L}} v_i$ *for each* e_i *then:*
  $\langle\!\langle \{\mathcal{E}_0, \lhd[\![P]\!]_{\text{M}} \rhd\}, \ldots, \{\mathcal{E}_{\boldsymbol{p_c}}, \lhd[\![P]\!]_{\text{M}} \rhd\} \rangle\!\rangle \Rightarrow^*$
  $\langle\!\langle \{\mathcal{E'}_0, \lhd v_0 \rhd\}, \ldots, \{\mathcal{E'}_{\boldsymbol{p_c}}, \lhd v_{\boldsymbol{p_c}} \rhd\} \rangle\!\rangle$
- *If* $\mathcal{M}, \vdash e_i \Downarrow_p^{\mathcal{L}} \infty$ *for one* e_i *then:*
  $\langle\!\langle \{\mathcal{E}_0, \lhd[\![P]\!]_{\text{M}} \rhd\}, \ldots, \{\mathcal{E}_{\boldsymbol{p_c}}, \lhd[\![P]\!]_{\text{M}} \rhd\} \rangle\!\rangle \Rightarrow^\infty$

Then, we get $\langle\langle e, ..., e \rangle\rangle \Rightarrow_{safe}$, *where* $\Rightarrow_{safe} \equiv \Rightarrow^* \cup \Rightarrow^\infty$.

The notation $\Downarrow_p^{\mathcal{L}}$ stands for the big step evaluation of a MULTI-ML expression on the processor p of locality $\mathcal{L}$ — see the rules in [5]. Then, $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$ denotes the evaluation of the expression e into the value v, within an environment of evaluation $\mathcal{M}$. Similarly, $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty$ denotes that the evaluation of e diverges. The generated code works similarly to the MULTI-ML's big step semantics. If all the expressions give values, then the code terminates giving a final value. Otherwise, if one expression diverges, then the whole machine is considered to be divergent.

2) Performances of the new implementation: As explained previously, the current implementation of the MULTI-ML language relies on MPI, with one process (daemon) for each component of the MULTI-BSP architecture. To run computations, a daemon sends function closures to be executed on its children. As the current implementation is SPMD, the code is known by all the processes and the function closures are not necessary. Thus, to execute code within the scope of a parallel vector, we just need to communicate a static identifier (standing for the function to be executed) in addition to the required values.

The amount of unnecessary communication can be quantified using an instrumented version of MULTI-ML which measures and sums the size of each function closure. To do so, we compute the size of the data transmitted at each communication step and we accumulate it in an global counter. As the function closures are merely small pieces of code, the communication overhead is quite low. We can observe, in Tab. I, the function closures overhead for several code examples: **T**ridiagonal **E**quation **S**ystems of equations (TDS) **F**ast **F**ourier **T**ransform (FFT);. And for some various architectures:

- i7, a multi-core with 2 cores and 2 threads per core;
- mirev3_1, a multi-core with 2 cores and 8 threads per core;
- mirev3, a cluster of 4 multi-cores with 2 cores and 8 threads per core;
- mirev3_{ht}, a cluster of 4 multi-cores with 2 cores and 16 threads per core;

Both examples use our implementation in MULTI-ML of a well-known data-parallel skeleton [10], [11] which is the **D**istributable **H**omomorphisms used to express divide-and-conquer algorithms.

As expected, the overhead is almost imperceptible for these small examples. We may also expect only a minor impact on bigger programs as the closures essentially contain *raw code* and as the number of closures is linear to the number of components of the MULTI-BSP architecture. The proposed implementation is no longer sensitive to sending data in a hidden way in some specific cases as explained above. As the generalisation relies on a direct code transformation, we do not expect any non-linear impact on the compilation time. This implementation works just as well but is not as sensitive to cost problems which is the main goal when programming algorithms with a *bridging model*.

IV. RELATED WORK

A. Abstract Machines

To calculate the values of the λ-calculus, a lot of abstract machines have been designed. The first was the SECD machine [12] which was used for the first implementation of the LISP language. There is also the CAM [13], the FAM [14] and many others. [15] introduced a powerful abstract machine, the ZAM, which underlies the byte-code interpreter of OCAML (using the ZAM2, unpublished). This machine was derived from the Krivine's abstract machine and from the λ-calculus with explicit subsections. This machine is interesting because its instructions could be "easily" translated into efficient byte-code (with some optimisations) and also to native code. It is to be noticed that for functional languages with a call by name strategy, [16] designs the G-machine with its graph reduction.

As we consider here an eager language, those techniques are not suitable for us.

For BSML, a first work has been done in [17] where the authors modified a SECD machine. However, this machine has two main drawbacks: (1) the number of processors of the machine which will execute the program has to be known at compilation; using an abstract machine eases portability but such a static definition of the number of processors is contradictory; (2) the exchange instructions for values is difficult to translate to real code, especially in an efficient way, as they add instructions to the code during the execution. These two problems were solved in [18]: only two new instructions have been added to a simple ZINC machine, one for having the processor identifier of computing unit and another one to perform a total exchange. The compilation scheme and the design of these two instructions have motivated the current modular implementation of BSML [2] (only a module of these 2 specific instructions need to be coded; different versions have been done such as MPI, BSPLIB, PVM, direct OCAML's TCP/IP routines, *etc.*).

Close to our work is the one of [19]. The authors define a distributed virtual machine for a lambda-calculus with streams (parallel computations over an infinite flow of data). However, this approach is not suitable for the MULTI-BSP.

B. Hierarchical programming

Many languages dedicated to hierarchical architectures were proposed. NESTSTEP [20] is a C/JAVA library for programming BSP algorithms. It allows nested computations on clusters of multi-cores. A data-parallel extension of HASKELL called NEPAL was proposed in [21], where an abstract machine is responsible for the distribution of the data over the available processors. We can also notice MULTIMLTON [22], which is a multi-core aware runtime for standard ML. This extension of the MLTON compiler manages composable and asynchronous events using safe-futures. LIFT [23] is a parallel pattern based high-level language targeting portability on parallel accelerators. An in-depth description of other bridging models for hierarchical architectures and other parallel languages can be found in [5]. We are currently not aware of dedicated languages that meets our requirements which are performance, portability and execution safety.

V. CONCLUSION AND FUTURE WORK

A. Conclusion

The compilation scheme presented in this article provides a detailed formal description of a distributed runtime system for MULTI-ML. It has two advantages compared to the past (and rather naive) implementation: (1) Only one process is needed on each computing unit (physical core); (2) It demonstrates the need for low level primitives to abstract the implementation in order to have a generic and modular implementation as in [2]. With such low-level primitives, it is easier to propose various implementations relying on different communication libraries, not limited to MPI. For example, we can imagine an optimised versions of the communication module for a specific architecture using a particular communication scheme.

This compilation scheme brings a generic approach allowing genericity – thanks to a limited set of primitives – and efficiency – using a close to hardware bridging model.

B. Future work

The next phases will be to: (1) Use the COQ proof assistant to get a machine-checked proof of correctness of this compilation scheme with respect to the semantics; that will give a greater confidence on the implementation (the compilation) of the MULTI-ML language; (2) Extend our work to be able to compile the full ML (OCAML) language with our parallel extension; For example, imperative features such as assignment and exceptions; This is an ongoing work: the main problem is to design an exception mechanism to deal with exceptions raised by the different daemons and thus handle various exceptions of a hierarchical architecture.

REFERENCES

[1] L. G. Valiant, "A Bridging Model for Parallel Computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[2] F. Loulergue, F. Gava, and D. Billiet, "Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction," in *Computational Science – ICCS 2005*. Springer, Berlin, Heidelberg, May 2005, pp. 1046–1054.

[3] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. [Online]. Available: http://www.springer.com/gp/book/9783540208549

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

[5] 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

[6] V. Allombert, F. Gava, and J. Tesson, "Multi-ML: Programming Multi-BSP Algorithms in ML," *International Journal of Parallel Programming*, vol. 45, no. 2, p. 20, Apr. 2017. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01160164

[7] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, "The OCaml system release 4.06: Documentation and user's manual," INRIA, Intern Report v4, Nov. 2017. [Online]. Available: https://hal.inria.fr/hal-00930213

[8] 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.

[9] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference*, 2nd ed. Cambridge, MA, USA: MIT Press, 1998.

[10] M. Cole, "Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming," *Parallel Comput.*, vol. 30, no. 3, pp. 389–406, Mar. 2004.

[11] M. H. Alt, "Using algorithmic skeletons for efficient grid computing with predictable performance," Ph.D. dissertation, Münster University, 2007.

[12] P. J. Landin, "The Mechanical Evaluation of Expressions," *The Computer Journal*, vol. 6, no. 4, pp. 308–320, Jan. 1964.

[13] G. Cousineau, P. L. Curien, and M. Mauny, "The Categorical Abstract Machine," in *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*. New York, NY, USA: Springer-Verlag New York, Inc., 1985, pp. 50–64. [Online]. Available: http://dl.acm.org/citation.cfm?id=5280.5284

[14] L. Cardelli, "Compiling a Functional Language," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. New York, NY, USA: ACM, 1984, pp. 208–217.

[15] X. Leroy, "The ZINC experiment: An economical implementation of the ML language," INRIA, Tech. Rep., 1990.

[16] S. L. Peyton Jones, *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.

[17] A. Merlin, G. Hains, and F. Loulergue, "An SPMD environment machine for functional BSP programs," 2001.

[18] F. Gava and F. Loulergue, "A Polymorphic Type System for Bulk Synchronous Parallel ML," in *Parallel Computing Technologies*. Springer, Berlin, Heidelberg, Sep. 2003, pp. 215–229.

[19] M. Pedicini, G. Pelliatta, and M. Piazza, "Sequential and Parallel Abstract Machines for Optimal Reduction," 2014.

[20] C. W. Kessler, "NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model," *The Journal of Supercomputing*, vol. 17, no. 3, pp. 245–262, Nov. 2000.

[21] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel, "Nepal - Nested Data Parallelism in Haskell," in *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*. London, UK, UK: Springer-Verlag, 2001, pp. 524–534. [Online]. Available: http://dl.acm.org/citation.cfm?id=646666.699740

[22] K. C. Sivaramakrishnan, L. Ziarek, and S. Jagannathan, "MultiMLton: A multicore-aware runtime for standard ML," *Journal of Functional Programming*, vol. 24, no. 06, pp. 613–674, 2014. [Online]. Available: http://journals.cambridge.org/article_S0956796814000161

[23] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: A Functional Data-parallel IR for High-performance GPU Code Generation," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 74–85. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049832.3049841