Toward performance prediction for Multi-BSP programs in ML

Victor Allombert, Frédéric Gava, Julien Tesson

To cite this version:

Victor Allombert, Frédéric Gava, Julien Tesson. Toward performance prediction for Multi-BSP programs in ML. 18th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), Nov 2018, Guangzhou, China. hal-01941250v1

HAL Id: hal-01941250
https://hal.science/hal-01941250v1

Submitted on 30 Nov 2018 (v1), last revised 10 Dec 2018 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Toward performance prediction for Multi-BSP programs in ML

Victor Allombert[1], Frédéric Gava[2], and Julien Tesson[2]

[1] Université d'Orléans, LIFO, Orléans, France
[2] Université Paris-Est Créteil, LACL, Créteil, France

Abstract. BSML and MULTI-ML are functional parallel programming languages "à la ML" based of the respectively the BSP and MULTI-BSP bridging models. MULTI-BSP extends BSP to take into account hierarchical architectures. For both models, it is possible to predict the performances of algorithms thanks to embedded cost models. To do so, we propose formal a operational semantics with cost annotations for the two aforementioned languages. This work has been done in a incremental manner. First we recall the cost semantics of core-ML language. Then, we adapt it to BSML and we adapt it to MULTI-ML. It is then possible to evaluate the cost of a program following the annotated semantics. Finally, we compare the theoretical approach with the current implementation on a code example.

Keywords: Semantics, BSP, BSML MULTI-BSP, Cost, Time prediction

1 Introduction

1.1 Context

The Culk Synchronous Parallelism (BSP) *bridging model* [19] was designed for *flat* parallel architectures. A bridging model is an abstract model of a computer which provides a conceptual bridge between the physical implementation of the machine and the *abstraction* available to a programmer of that machine. But modern High Performance Computing (HPC) architectures are now *hierarchical* and have multiple layers of parallelism, communication between distant nodes cannot be as fast as among the cores of a given processor. We now consider the MULTI-BSP model [20], an extension of BSP. MULTI-ML [?] is a MULTI-BSP extension of BSML [9], a functional approach for programming BSP algorithms in ML, BSML being itself an extension of OCAML, a ML language (`https://ocaml.org/`).

To be compliant with a bridging model eases the way of writing codes that ensures *efficiency* and *portability* from one architecture to another and also avoid deadlocks and non-determinism. The MULTI-BSP bridging model offers a high level of abstraction and takes into account real communications and synchronisation costs on hierarchical architectures. Thanks to the cost model embedded in the (MULTI-)BSP model, it is possible obtain the cost of a given algorithm. Using the (MULTI-)BSP parameters of an architecture allows to predict the execution time of a given code. That can useful for resource bound analysis and find performance bugs thus to provide development-time feedback to HPC programmers.

We chose OCAML (with our own distributed extensions) as the source language "à la ML" for several reasons. For one, OCAML is a widely used language for functional programming which is quite efficient in practice (sophisticated compiler and automatic memory management). Moreover, we wanted to demonstrate that it is possible to define a practical cost semantics for high-level HPC languages; imperative programming is closer to standard assembly codes which already have their cost analysis such as WCET [**?**]. Even if functional programming is currently not the norm for HPC, it is more and more common that main stream languages (such as JAVA) add functional features. Studying these features in ML, without having to manage others features (such as JAVA's objects), is a classical manner to get them for other languages.

Cost prediction is important for the design of efficient algorithms and is also important in domains where programs are executed with time constraints (such as in physical engines such as aeroplanes *etc.*). In the future, even such domains would benefit of many-cores architectures (at most). Cost prediction of HPC programs is thus an important issue to ensure the safety of such systems.

1.2 Example of the methodology: the sequential case

An important first step to study cost prediction of programs is to define the cost of the construction of the language itself, that is define an operational big-step semantics that assign a parametric cost to a well-formed expression. Having a compositionnal cost semantics is also an important issue in order to get modular and incremental programming: from a software engineering point of view, it makes senses that the cost of a subprogram does not depend (too much) on the context, for example, the cost of an array sorting method should depend only on the size of the input and not when it is called.

The main hypothesis is that the resource consumption of a program is a linear combination of the number of executions of each construct in the program[3]. The semantics models this idea by parameterizing the cost with unknown coefficients that correspond to each ML construct: the number of executions of each of these constructs constitutes the majority of the execution time of most ML programs.

Taking the case of the core-ML language. It relies on a minimal set of ML constructions. This set is sufficient enough to express all the behaviour that are used in ML programming. Thus, features such as records, modules, pattern matching, sum types are excluded. The grammar is:

$e ::=$	$\mathbf{cst}$	*Constants*	$\mid$	$\mathbf{let}\ x = e\ \mathbf{in}\ e$	*Binding*
$\mid$	$\mathbf{op}$	*Operators*	$\mid$	$\mathbf{fun}\ x \to e$	*Function*
$\mid$	x	*Variables*	$\mid$	$\mathbf{rec}\ f\ x \to e$	*Recursive function*
$\mid$	$(e\, e)$	*Application*	$\mid$	$\mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e$	*Conditional*

In this grammar, `x` and f range over an infinite set of *identifiers*. We also find the typical ML-like constructors such as **let** for bindings and also `fun` and `rec` for, respectively, functions and recursive functions. As expected, the application

[3] But their combination could be not linear as for algorithms with polynomial or exponential complexities.

is denoted $(e\ e)$. For the sake of readability, we take the liberty to use the familiar infix notation for binary operators, as well as the usual precedence and associativity rules. When the context is clear, we can avoid the usage of parentheses. **op** stands for the standard operators, such as common computations on integers. **cst** stands for constants such as integers, booleans, *etc.* An expression is evaluated into a value v which are defined as:

$$\texttt{v} ::= \mathbf{op} \quad | \quad \mathbf{cst} \quad | \quad \overline{(\mathbf{fun}\ x \to e\ )[\mathcal{E}]} \quad | \quad \overline{(\mathbf{rec}\ f\ x \to e\ )[\mathcal{E}]}$$

$$\mathcal{E} ::= \{x_1 \mapsto v_1, \ldots x_n \mapsto x_n\}$$

Values contains constants and closures (a value which stores both a function and its environment). An environment $\mathcal{E}$ is interpreted as a partial mapping with finite domain from identifiers to values. The extension of $\mathcal{E}$ by v in x is written $\mathcal{E} \uplus \{x \mapsto v\}$. An inference rule can be written as following:

$$\frac{\mathcal{P}}{\mathcal{E} \vdash e \Downarrow v \rightsquigarrow C}$$

That is with the premise $\mathcal{P}$, the expression e is evaluated to the value v at cost C. The cost (time and memory) consumed by each construct is averaged out to be a constant. Hence, the execution time of a program C is: $\sum_{\mathbf{c}\in\mathcal{C}} n_{\mathbf{c}} \times T_{\mathbf{c}}$ where $\mathcal{C}$ represents the set of constructs and $n_{\mathbf{c}}$ is the count of each construct during the whole program execution, and $T_{\mathbf{c}}$ is the execution time of the respective constructs. Estimating the overall time execution of a program (in "seconds") from the semantics now consists to estimating each $T_{\mathbf{c}}$ (in μs) using micro-benchmarking[4] and replacing them into the extracted cost C. The inference rules for core-ML are defined in Fig. 1 and work as follow.

The Csts and Ops rules do not generate any additional cost. Indeed, we assume that they are static values which are accessible freely. Vars aims to access a value bound in a memory using the *lookup* operator (which returns the corresponding bound value). As this operator access a value stored in a memory, its cost should be proportional to the path trough different caches-memories. However, we chose to set such a constant T_{var} in order to simplify the rules.

The Closure rule mainly models the way the values are enclosed inside a function closure. It is done using the *select* operator which, given an environment $\mathcal{E}$ and a function (code) returns the minimal environment to evaluate such a code. We assume that the cost of building such an environment is proportional to the number of free variables ($\mathcal{F}$, define by trivial induction on expressions) of e. It is an approximation which can be refined by taking into account more OCAML mechanisms. Recursive functions are build in the same way.

The APP, LET and IF rules are straightforward: we simply propagate the cost produced by each expressions. Note the modification of the environment for the application to evaluate the code of the closure. Also, each operator gets a cost noted c_3 in the rule and we note $\overline{\mathbf{op}\ v}$ the new built value. The "s" on the rules that are unused here but will be necessary for the BSP's supersteps. It is also straightforward to show that $\oplus$ is commutative.

[4] This assumption does not truly holds for most of the relevant platforms (*e.g.* the garbage collector and caches-misses) but is still sufficient for our study; We let more subtle analyses to future works and we will focus on parallelism.

$$\text{CSTS}\quad \frac{}{\mathcal{E} \vdash^s \mathbf{cst} \Downarrow \mathbf{cst} \rightsquigarrow^s 0} \qquad \text{OPS}\quad \frac{}{\mathcal{E} \vdash^s \mathbf{op} \Downarrow \mathbf{op} \rightsquigarrow^s 0} \qquad \text{VARS}\quad \frac{\{x \mapsto v\} \in lookup(x, \mathcal{E})}{\mathcal{E} \vdash^s x \Downarrow v \rightsquigarrow^s T_{var}}$$

$$\text{CLOSURES}\quad \frac{\mathcal{E}' = select(\mathcal{E}, \mathcal{F}(\mathbf{fun}\ x \rightarrow e\ )) \rightsquigarrow n = |v| \qquad v \equiv \overline{(\mathbf{fun}\ x \rightarrow e\ )[\mathcal{E}']}}{\mathcal{E} \vdash^s \mathbf{fun}\ x \rightarrow e\ \Downarrow v \rightsquigarrow^s T_{def} \oplus n \times T_{clo}}$$

$$\text{APP1}\quad \frac{\mathcal{E} \vdash^s e_1 \Downarrow \overline{(\mathbf{fun}\ x \rightarrow e_3\ )[\mathcal{E}']} \rightsquigarrow^{s_1} c_1 \quad \mathcal{E} \vdash^{s_1} e_2 \Downarrow v \rightsquigarrow^{s_2} c_2 \quad \mathcal{E}' \uplus \{x \mapsto v\} \vdash^{s_2} e_3 \Downarrow v' \rightsquigarrow^{s_3} c_3}{\mathcal{E} \vdash^s (e_1\ e_2) \Downarrow v' \rightsquigarrow^{s_3} c_1 \oplus c_2 \oplus c_3 \oplus T_{FunApp}}$$

$$\text{APP2}\quad \frac{\mathcal{E} \vdash^s e_1 \Downarrow \mathbf{op} \rightsquigarrow^{s_1} c_1 \quad \mathcal{E} \vdash^{s_1} e_2 \Downarrow v \rightsquigarrow^{s_2} c_2 \quad v' \equiv \overline{\mathbf{op}\ v} \rightsquigarrow c_3}{\mathcal{E} \vdash^s (e_1\ e_2) \Downarrow v' \rightsquigarrow^{s_2} c_1 \oplus c_2 \oplus c_3}$$

$$\text{LET}\quad \frac{\mathcal{E} \vdash^s e_1 \Downarrow v_1 \rightsquigarrow^{s_1} c_1 \quad \mathcal{E} \uplus \{x \mapsto v\} \vdash^{s_1} e_2 \Downarrow v_2 \rightsquigarrow^{s_2} c_2}{\mathcal{E} \vdash^s \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow v_2 \rightsquigarrow^{s_2} c_1 \oplus c_2 \oplus T_{let}}$$

$$\text{IF1}\quad \frac{\mathcal{E} \vdash^s e_1 \Downarrow \texttt{True} \rightsquigarrow^{s_1} c_1 \quad \mathcal{E} \vdash^{s_1} e_2 \Downarrow v_2 \rightsquigarrow^{s_2} c_2}{\mathcal{E} \vdash^s \mathbf{if}\ e_1 \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v_2 \rightsquigarrow^{s_2} c_1 \oplus c_2 \oplus T_{if}}$$

$$\text{IF2}\quad \frac{\mathcal{E} \vdash^s e_1 \Downarrow \texttt{False} \rightsquigarrow^{s_1} c_1 \quad \mathcal{E} \vdash^s e_3 \Downarrow v_3 \rightsquigarrow^{s_3} c_3}{\mathcal{E} \vdash^s \mathbf{if}\ e_1 \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v_3 \rightsquigarrow^{s_3} c_1 \oplus c_3 \oplus T_{if}}$$

Fig. 1. The cost semantics of the sequential core-ML language.

1.3 Outlines

In this article we introduce the formal cost semantics of first the BSML (Section 2) and then we extend it to MULTI-ML (Section 3). For both languages, we first present the model of execution, then the cost model and we give the semantics annotated with costs for core languages that describes the syntax of the aforementioned languages. Finally, we compare the predicted execution times with the actual one on a small example (Section 4). After the description of related works (Section 5) we conclude (Section 6).

2 BSP programming in ML and costs semantics

2.1 The BSP bridging model

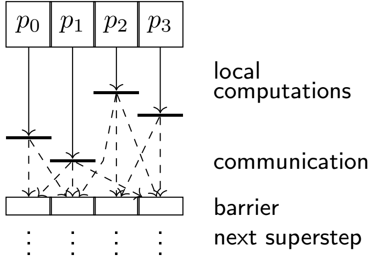

Fig. 2. A BSP superstep.

In the BSP model [19], a computer is a set of $\mathbf{p}$ *uniform* pairs of processor-memory with a communication network. A BSP program is executed as a *sequence* of *supersteps* (Fig. 2), each one divided into three successive disjointed phases: (1) each processor only uses its local data to perform sequential computations and to request data transfers to other nodes; (2) the network delivers the requested data; (3) a *global synchronisation barrier* occurs, making the transferred data available for the next superstep.

As BSP architecture can be easily mapped on any general purpose parallel architecture. Thanks to the BSP cost model it is possible to accurately *estimate* the execution time of a BSP program with the BSP parameters. The performance of a

BSP computer is characterised by four parameters: The local processing speed **r**; The number of processors **p**; The time **L** required for a barrier; The time **g** for collectively delivering a 1-relation. **g** and **L** can be expressed in FLoating-point Operations (FLOPS) and **r** in FLOPS per second. To accurately *estimate* the execution time of a BSP program, these 4 parameters can be easily benchmarked [3].

A 1-relation is a collective exchange where every processor receives/sends at most one word. The network can deliver an h-relation in time $\mathbf{g} \times h$. The execution time (cost) of a superstep s is the sum of the maximal local processing time, the data delivery and the global synchronisation times. It is expressed by the following formula: $\text{Cost}(s) = \max_{0 \leq i < \mathbf{p}} w_i^s + \max_{0 \leq i < \mathbf{p}} h_i^s \times \mathbf{g} + \mathbf{L}$ where w_i^s is the local processing time on processor i during superstep s and h_i^s is the maximal number of words transmitted or received by processor i during superstep s. The total cost of a BSP program is the sum of its supersteps's costs.

2.2 The BSML language

BSML [8] uses a *small set of primitives* and is currently implemented as a library (`http://traclifo.univ-orleans.fr/bsml/`) for the ML programming language OCAML. An important feature of BSML is its *confluent* semantics: whatever the order of execution of the processors is, the final value will be the same. Confluence is convenient for *debugging* since it allows to get an *interactive loop* (toplevel). That also simplifies programming since the parallelisation can be done *incrementally* from an OCAML program.

A BSML program is built as a ML one but using a specific data structure called *parallel vector*. Its ML type is 'a par. A vector expresses that each of the **p** processors *embeds* a value of any type 'a. Fig. 3 resumes the BSML primitives. Informally, they work as follows: let $\ll$ `e` $\gg$ be the vector holding `e` everywhere (on each processor), the $\ll \ \gg$ indicates that we enter into the scope of a vector. Within a vector, the syntax `$x$` can be used to read the vector `x` and get the local value it contains. The *ids* can be accessed with the predefined vector **pid**. When a value is referenced within the scope of a parallel vector, its *locality* is `l` (local); otherwise, the locality is `b` (BSP).

The **proj** primitive is the only way to *extract* local values from a vector. Given a vector, it returns a function such that applied to the *pid* of a processor, returns the value of the vector at this processor. **proj** performs communication to make local results available globally and ends the current superstep.

The **put** primitive is another communication primitive. It allows any local value to be *transferred* to any other processor. It is also synchronous, and ends the current superstep. The parameter of **put** is a vector that, at each processor, holds a function returning the data to be sent to processor j when applied to j. The result of **put** is another vector of functions: at a processor j the function, when applied to i, yields the value *received from* processor i by processor j.

Primitive	Type	Informal semantics
$<< e >>$	'a par (if e:'a)	$\langle e, \ldots, e \rangle$, a vector of size $\mathbf{p}$ the number of processors
pid	int par	A predefined vector: i on processor i
\$v\$	'a (if v: 'a par)	v_i on processor i, assumes $v \equiv \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$
proj	'a par−> (int−>'a)	$\langle x_0, \ldots, x_{\mathbf{p}-1} \rangle \mapsto (\text{fun } i \to x_i)$
put	(int −> 'a)par−> (int−> 'a)par	$\langle f_0, \ldots, f_{\mathbf{p}-1} \rangle \mapsto \langle (\text{fun } i \to f_i\ 0), \ldots, (\text{fun } i \to f_i\ (\mathbf{p}-1)) \rangle$

Fig. 3. Summary of the BSML primitives.

2.3 Cost semantics

Extension. To obtain core-BSML, we extends the expressions of core-ML with parallel primitives as follow: $e ::= \cdots \mid$ `replicate` (**fun** _ $\to e$) $\mid$ (`proj` e) $\mid$ (`put` e) $\mid$ (`apply` $e\,e$). The distinction made between the syntactic sugar (the $\ll\ \gg$ and `$` notations), used when programming BSML algorithms, and the core parallel primitives (`replicate` and `apply`), available in the semantics only, simplifies the semantics. Indeed, the syntactic sugar eases the way of programming but it is not suitable for the semantics as it introduces implicit assumptions. Thus, we must transform and abstract the syntactic sugar using the *core parallel primitives*. The transformation applied to switch from the syntactic sugar to the core parallel primitives is straightforward and produce and equivalent expression. The parallel vector scope, denoted $\ll$ e$\gg$, is transformed using the `replicate` core primitive. Thus, $\ll$ e$\gg$ is simply transformed into `replicate` (**fun** _ $\to e$). The \$ syntax is transformed using the `apply` primitive. The transformation is simple and does not require a complicated expression analysis. To do so, we build a vector of functions that takes, as argument, the dollar's annotated value. Using the `apply` primitive, we can *apply* this vector of functions on the vector of values. For example, the expression $\ll$ (e \$x\$) $\gg$ is transformed into `apply` (`replicate` (**fun** _ $x \to e\ x$)) x.

Values are also extended with parallel vectors: $v ::= \cdots \mid\ < v_1, \ldots, v_{\mathbf{p}} >$. In the following, to simplify the notations, we indices processors from 1 to $\mathbf{p}$ (and not from 0 to $\mathbf{p}-1$ as common in HPC). We make also the hypothesis that there exists a special vector named pid$=< 1, \cdots, \mathbf{p} >$ (the ids of the processors).

The main modification is about the costs. During a superstep, the asynchronous costs are counting independently and it is only during the barrier that the maximal of the costs (computation and communication) are to be taken into account. But a same superstep can be in two different parts of an expression (for example **let** x=$\ll$ 1+1$\gg$ **in** ((**proj** $\ll$ \$x\$+1$\gg$) 2) where the begin of the first superstep is in the first part of the **let**, the next just before the call of the **proj** and the second superstep when apply the result of the **proj** on the constant 2). For this reason, we extends the costs with vector of costs $< c_1, \ldots, c_{\mathbf{p}} >_s$ where each component i describe the current local cost c_i of processor i during the superstep s. This s is modify only by the rules of synchronous primitives. Nevertheless, we add the three following equivalences:

1. $< c_1, \ldots, c_{\mathbf{p}} >_s \oplus < c'_1, \ldots, c'_{\mathbf{p}} >_s \equiv < c_1 \oplus c'_1, \ldots, c_{\mathbf{p}} \oplus c'_{\mathbf{p}} >_s$, if c_i and c'_i does not contains vectors
2. $< T_{op} \oplus c_1, \ldots, T_{op} \oplus c_{\mathbf{p}} >_s \equiv T_{op} \oplus < c_1, \ldots, c_{\mathbf{p}} >_s$, whatever T_{op}
3. $0 \equiv < 0, \ldots, 0 >_s$, whatever s

These rules aims to keep using the previous rule of the sequential constructions of the languages (let, fun, *etc.*).

$$\textsc{Rpl}\ \frac{\forall i \in \{1,\ldots,\mathbf{p}\} \quad \mathcal{E} \vdash^s e \Downarrow v_i \leadsto^s c_i \qquad \text{if } Valid(e,\mathcal{E})}{\mathcal{E} \vdash^s \texttt{replicate}\,(\textbf{fun}\ _ \to e\ ) \Downarrow < v_1,\ldots,v_{\mathbf{p}} > \leadsto^s T_{rpl} \oplus < c_1,\ldots,c_{\mathbf{p}} >_s}$$

$$\textsc{Apply}\ \frac{\begin{cases} \mathcal{E} \vdash^s e_1 \Downarrow < f_1,\ldots,f_{\mathbf{p}} > \leadsto^{s_1} c_1 \\ \mathcal{E} \vdash^{s_1} e_2 \Downarrow < v_1,\ldots,v_{\mathbf{p}} > \leadsto^{s_2} c_2 \end{cases} \quad \forall i \in \{1,\ldots,\mathbf{p}\} \quad \mathcal{E} \vdash^{s_2} (f_i\ v_i) \Downarrow v'_i \leadsto^{s_2} c'_i}{\mathcal{E} \vdash^s (\texttt{apply}\, e_1\, e_2) \Downarrow < v'_1,\ldots,v'_{\mathbf{p}} > \leadsto^{s_2} T_{app} \oplus c_1 \oplus c_2 \oplus < c'_1,\ldots,c'_{\mathbf{p}} >_{s_2}}$$

$$\textsc{Proj}\ \frac{\mathcal{E} \vdash^s e \Downarrow < v_1,\ldots,v_{\mathbf{p}} > \leadsto^{s'} c \quad f \text{ is such that } \forall i \in \{1,\ldots,\mathbf{p}\} \ \ \mathcal{E} \vdash (f\ i) \equiv v_i}{\mathcal{E} \vdash^s (\texttt{proj}\, e) \Downarrow f \leadsto^{s'+1} T_{proj} \oplus c \oplus HRelation(v_1,\ldots,v_{\mathbf{p}}) \times \mathbf{g} \oplus \mathbf{L}}$$

Fig. 4. The cost semantics of the core-BSML language.

Lemma 1. *The costs with parallel vector of costs form a commutative and associative group id where* 0 *is the neutral element inside or outside cost vectors and where* $< 0,\ldots,0 >_s$ *is the neutral element outside vectors only.*

Adding rules. We must now extend our inference rules in order to take into account the BSP primitives. These rules are given in Fig 4. They work as follow.

The Rpl rule is for building asynchronously a new parallel vector. The expression e is evaluated for each component, in parallel, making a new vector of cost for the current superstep s. The valid function is used to forbid nested vectors and is fully defined in [1]. A type system has been designed to not be forced to do this check dynamically. Then a construct is linearly add.

The Apply rule works similarly but for two expressions which thus add two different costs (not necessary vectors and for possibly different supersteps) and we finally built the vector by computing its components in parallel (on each processor) making the linear add of a new costs vector.

The Proj rule adds a barrier ($\mathbf{L}$) and thus finishes the superstep (updating s). From the exchanged computing values, a h-relation is added: $\mathbf{g}$ and $\mathbf{L}$ are thus special constructs. The PUT cost is quite dense because of the number of communications between all the processors which are done during the evaluation of the primitive. But the rule is close the `proj` one. For sake of conciseness, we do not show it. The way the data sizes are computed by simple induction on the values ($Hrelation$): it is rather naive but sufficient to an upper born.

To get the overall execution time $\mathcal{E} \vdash^s e \Downarrow v \leadsto^{s'} c$ then it is $\max(c) \oplus \mathbf{L}$ where the function max first apply the three previous equivalences in order to aggregate (merge) the cost vectors of the same superstep until not merging is possible. Finally, when the cost (time and memory) consumed by each construct is statically known in μs then $\max(< c_1,\ldots,c_{\mathbf{p}} >_s) = c_i$ if $\forall j \neq i, c_j \leq c_i$.

Lemma 2. *max is idempotent that is* $\forall c\ \ \max(\max(c)) = \max(c)$.

For example, `let x=<<1+1>> in ((proj <<$x$+1>>) 2)` beginning with whatever environment $\mathcal{E}$ at any superstep s, for a two processors BSP machine, the cost semantics indicates that the adding cost of such expression is: $< T_+, T_+ >_s \oplus T_{rpl} \oplus T_{app} \oplus < T_{var} \oplus T_+, T_{var} \oplus T_+ >_s \oplus 1 \times \mathbf{g} \oplus \mathbf{L} \oplus T_{app}$ (2 vectors constructions both with an addition; a synchronous primitive; and a final application). That is to say, in any context, the expression adds T_+ during the asynchronous phase of the current superstep s, finishes it and begins a new superstep. On it own, the cost of such an expression can be simplify into $2 \times T_+ \oplus \mathbf{g} \oplus \mathbf{L}$.

3 Multi-BSP programming in ML and costs semantics

3.1 The Multi-BSP bridging model

MULTI-BSP is a bridging *model* [20] which is adapted to hierarchical architectures, mainly *clusters* of *multi-cores*. It is an extension of the BSP bridging model. The structure and abstraction brought by MULTI-BSP allows to have portable programs with *scalable* performance predictions, without dealing with low-level details of the architectures. This model brings a *tree*-based view of nested components (*sub-machines*) of hierarchical architectures where the lowest stages (*leaves*) are processors and every other stage (*nodes*) contains memory.

Every component can execute code but they have to synchronise in favour of data exchange. Thus, MULTI-BSP does not allow subgroup synchronisation of any group of processors: at a stage i there is only a synchronisation of the sub-components, a synchronisation of each of the computational units that manage the stage $i-1$. So, a node executes some code on its nested components (*aka* "*children*"), then waits for results, does the communication and synchronises the sub-machine. A MULTI-BSP algorithm is thus composed by several supersteps, each step is synchronised for each sub-machine.

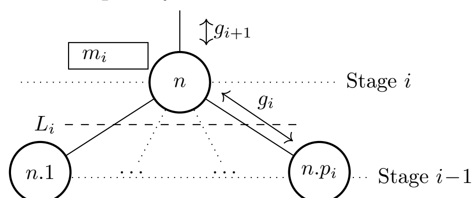

Fig. 5. The Multi-BSP parameters

An instance of MULTI-BSP is defined by $\mathbf{d}$, the fixed depth of the (balanced and homogeneous) tree architecture, and by 4 parameters for each *stage* i of the tree : $(\mathbf{p_i}, \mathbf{g_i}, \mathbf{L_i}, \mathbf{m_i})$; described in Fig. 5: $\mathbf{p_i}$ is the number of sub-components inside the $i-1$ stage; $\mathbf{g_i}$ is the *bandwidth* between stages i and $i-1$: the ratio of the number of operations to the number of words that can be transmitted in a second; $\mathbf{L_i}$ is the *synchronisation cost* of all sub-components of a component of $i-1$ stage; $\mathbf{m_i}$ is the amount of memory available at stage i for each component of this stage.

Thanks to those parameters, the cost of a MULTI-BSP algorithm can be computed as the sum of the costs of the supersteps of the root node, where the cost of each of these supersteps is the maximal cost of the supersteps of the sub-components (plus communication and synchronisation); And so on.

Let C_j^i be the communication cost of a superstep j at stage i: $C_j^i = h_j \times \mathbf{g}_i + \mathbf{L}_i$ where h_j the maximum size of the exchanged messages at superstep j, $\mathbf{g}_i$ the communication bandwidth with stage i and $\mathbf{L}_i$ the synchronisation cost. We can express the cost T of a MULTI-BSP algorithms as following:

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

where $\mathbf{d}$ is the depth of the 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.

It is to notice that the BSP and MULTI-BSP cost models both are a linear combination of costs for the asynchronous computations and costs of communications (separated by barriers).

3.2 The Multi-ML language

MULTI-ML [?,1] (`https://git.lacl.fr/vallombert/Multi-ML`) is based on the idea of executing BSML-like codes on every stage of a MULTI-BSP architecture. This approach facilitates *incremental* development from BSML codes to MULTI-ML ones. 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, as in BSML. This parallel data structure aims to manage values that are stored on the sub-nodes: at stage i, the code **let** v=≪ e≫ evaluates the expression e on each $i-1$ stages.

Inside a vector, we note #x# to copy the value x stored at stage i to the memory $i-1$. The (**mkpar** f) primitive is an alternative way to build a vector using a function f. Typed (int $\to \alpha) \to \alpha$ par, it aims to execute the given function to each processor identifiers (from 0 to $p_i - 1$) of a node locally on it; and then, distribute the results down to its sub-nodes. The main difference with the ≪ e≫ notation is that (**mkpar** f) aims to reduce costs when the communication costs of e is high and the execution cost of f and its result is low. As in BSML, we also found the **proj**, **put** primitives and the syntax x, all of them with the same semantics.

We also introduce 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 (and ends) on the root node. The following code shows how a multi-function is defined.

```
let multi mf [args]=
 | where node =
 | | (* BSML code*)
 | | ...
 | | ≪ mf [args] ≫
 | | ... in v
 | where leaf =
 | | (* OCaml code *)
 | | ... in v
```

After the definition of the multi-function mf on line 1 where [args] symbolises a set of arguments, we define the node code (from line 2 to 6). The recursive call of the multi-function is done on line 5, within the scope of a parallel vector. The node code ends with a value v, which is available as a result of the recursive call from the upper node. The leaf code, from lines 7 to 9 consists of sequential computations.

We also propose another parallel data structure called *tree*. A tree is a distributed structure where a value is stored in every nodes and leaves memories. A tree can be built using a multi-tree-function, with the **let multi tree** keyword and can be handled by several primitives of the language. We do not detail this construction here.

Similarly to BSML and its `b` and `l` localities, in MULTI-ML we introduce `m` when a value refers to the MULTI-BSP locality and `s` on leaves (sequential).

3.3 Cost semantics

Extension. To obtain core-MULTI-ML, we extends core-BSML with multi-functions as follow: $e ::= \cdots \mid (\texttt{down}\, x) \mid \textbf{multi}\ f\ x \to e \dagger e$.

$$\text{MultiNode}\quad \frac{\begin{cases}\mathcal{E} \vdash^{s} e_1 \Downarrow^{l}_{n} \overline{(\textbf{multi}\ f\ x \rightarrow e'_1 \dagger e'_2)[\mathcal{E}']} \rightsquigarrow^{s_1} c_1 \\ \mathcal{E} \vdash^{s_1} e_2 \Downarrow^{l}_{n} v \rightsquigarrow^{s_2} c_2\end{cases} \quad \mathcal{E}' \vdash^{0} e'_1 \Downarrow^{b}_{n+1} v' \rightsquigarrow^{s_3} c_3}{\mathcal{E} \vdash^{s} (e_1\ e_2) \Downarrow^{l}_{n} v' \rightsquigarrow^{s_3} T_{app} \oplus c_1 \oplus c_2 \oplus \max(c_3) \oplus \mathbf{L}_n)}$$

$$\text{MultiLeaf}\quad \frac{\begin{cases}\mathcal{E} \vdash^{s} e_1 \Downarrow^{l}_{n} \overline{(\textbf{multi}\ f\ x \rightarrow e'_1 \dagger e'_2)[\mathcal{E}']} \rightsquigarrow^{s_1} c_1 \\ \mathcal{E} \vdash^{s_1} e_2 \Downarrow^{l}_{n} v \rightsquigarrow^{s_2} c_2\end{cases} \quad \mathcal{E}' \vdash^{s_2} e'_2 \Downarrow_{n+1} v' \rightsquigarrow^{s_2} c_3}{\mathcal{E} \vdash^{s} (e_1\ e_2) \Downarrow^{l}_{n} v' \rightsquigarrow^{s_2} T_{app} \oplus c_1 \oplus c_2 \oplus c_3)}$$

$$\text{MultiCall}\quad \frac{\begin{cases}\mathcal{E} \vdash^{s} e_1 \Downarrow^{m} \overline{(\textbf{multi}\ f\ x \rightarrow e'_1 \dagger e'_2)[\mathcal{E}']} \rightsquigarrow^{s} c_1 \\ \mathcal{E} \vdash^{s} e_2 \Downarrow^{m} v \rightsquigarrow^{s} c_2\end{cases} \quad \mathcal{E}' \vdash^{0} e'_2 \Downarrow^{b}_{1} v' \rightsquigarrow^{s_3} c_3 \quad Valid(v', \mathcal{E}')}{\mathcal{E} \vdash^{s} (e_1\ e_2) \Downarrow^{m} v' \rightsquigarrow^{s} T_{app} \oplus c_1 \oplus c_2 \oplus \max(\max(c_3)) \oplus SizeOf(v') \times \mathbf{g} \oplus \mathbf{L}}$$

Fig. 6. The cost semantics of the core-MULTI-ML language.

The multi-function definition is written with the keyword **multi**. It takes one arguments and two expressions separated by the † symbol; the first argument stands for the node code and the second is for leaf code. The `down` primitive aims to transfer a value to all the sub-nodes. The transformation from the # syntax into the `down` primitive is obvious and work as other syntactic sugars of BSML. For example, the expression $<< e\ \#x\#\ >>$ is transformed into `apply` (`replicate` (**fun** _ $x \rightarrow e\ x$))(`down` x). As the # annotated value is given as argument of the vector of functions, there are no redundant copies. The expression $<< \#x\# + \#x\#\ >>$ is transformed into a code that copy x to the sub-nodes, only once. Parallel vectors of values (and costs) now also depend of their deep level n in the MULTI-BSP architecture. Closures of multi-functions are also added. Thus we have $v ::= \cdots \mid\ < v_1, \ldots, v_{\mathbf{p}_n} > \ \mid \overline{(\textbf{multi}\ f\ x \rightarrow e \dagger e)[\mathcal{E}]}$.

Adding rules. We must now extend our inference rules in order to take into account the multi-functions and the nested BSML codes. These rules are given in Fig. 6. They work as follow.

These new rules need some updates of the previous rules. First, the $\Downarrow$ is parameterized by the different levels of execution of MULTI-ML and the stage n (beginning from 1). BSML rules has to be trivially updated with this stage in order to build the right size vectors.

As a node is a particular component where it is possible to express BSP parallelism, we must consider the synchronous costs generated by BSP computations. Those rules, at a stage n, are used to recurse trough the MULTI-BSP architecture using the multi-function. Therefore, the max function now first merge the vectors of the same (sub)superstep and finally we use this following equivalence (for each superstep s): $\max(n_1 \times T_1 \oplus \cdots \oplus n_t \times T_m \oplus < c_1, \ldots, c_{\mathbf{p}_n} >_s) \equiv \max(n_1 T_1 \oplus \cdots \oplus n_t \times T_t, \max_{i=1..\mathbf{p}_n}(c_i))$ that is we take the maximum between the computation of the node parent with the max of its own children.

The MULTICALL rule is for calling the multi-function at the level m. The counter of superstep is initiated to 0 as the stage to 1. The code of the node begins (level b). This rule terminates with a whole and synchronous broadcasting of the final value v where $\mathbf{g} = \mathbf{g}_1 + \mathbf{g}_2 ... + \mathbf{g}_d$ (as well for $\mathbf{L}$); This is due to the

model of execution of MULTI-ML where the code outside multi-function is run by all the processors in order to manage the whole execution and thus the value must be known by all the processors. The maximum function allow to get the right cost of all child. The rule is possible only if v is valid (as in BSML). Our type system forbids expressions that have not this property [1] and we can assume that all the evaluated expressions are correct.

The MULTILEAF goes to leaf level. The number of supersteps still the same when going throw the leaf level (only sequential codes are allow).

The MULTINODE is for going throw the hierarchical architecture (inside a vector) from one node to another one (the child). Thus the stage is incremented. A final synchronisation is used to finally wait all the child before terminating the node code (the recursive call of the multi-function). This allow to take the maximum of computation of the sub supersteps as wanted in the MULTI-BSP cost model. In MULTI-ML, the building of a vector is an asynchronous operation with a emission of a signal of creation from the node processor to the subnodes (or leaves). It is thus no longer possible using the second equivalence of the $\oplus$ which only becomes commutative between two $\mathbf{L_n}$ (barrier) at a stage n.

It is to notice that the *Lookup* function need also to check the variable at the right memory. Indeed, a variable define in at the stage n is no available on another stages. To do this, one must adding indices in the environment $\mathcal{E}$. More details are available in [1].

Here, only the MULTINODE and MULTILEAF rules can be evaluated. The costs of the multi-function recursive call taking place on both the node and the leaf is simple. We just add the evaluation cost of e_1 and e_2, plus the multi-function call cost, resulting in the recursive call. The MULTINODE rule adds the C_i costs which result from the potential asynchronous computations done on the node. Thus, we collect all the costs engendered by multi-function recursion. As expected, this mechanism is not necessary on the MULTILEAF rule, as there is no parallel computation at this level.

4 Experiments

Thanks to the cost model embedded in the MULTI-BSP model, it is possible to estimate the evaluation cost of a MULTI-ML program. According to the MULTI-BSP parameters standing for a machine specification, it is then possible to predict the execution time of a program.

To verify that the cost estimation retrieved from the MULTI-BSP cost formulae is valid, we are going to compare the computation time of a simple algorithm to the predicted computation cost. To do so, we propose to analyse a matrix vector product algorithm based on the map/reduce skeleton. Using the MULTI-BSP parameters of the targeted architecture able to predict the computation time of various inputs. Our example has been written in a functional style using tail-recursive functions but thanks to the OCAML compiler, these functions are transformed into an efficient imperative version.

4.1 Algorithm description

We consider a simple algorithm to compute the product of a matrix and a vector. Given a matrix $\mathcal{M}$ of dimension $n \times m$, where n stands for the number of lines and m form the number of columns, and a vector $\mathcal{V}$ of dimension n (number of lines) the computation is the following: $\mathcal{M} \times \mathcal{V} = x$, such as $x = (x_0, \ldots, x_n)$ where x is composed by m lines and $x_i = \sum_{j=0}^{n} \mathcal{M}_{ij} \times \mathcal{V}_j$. Now, to propose a parallel version of this matrix vector product, we choose to use the map/reduce skeleton [6]. Using map/reduce algorithms is an easy way to propose parallel algorithms using simple associative and commutative operators. A map/reduce algorithms works as following: (1) the data are distributed among the processing units; (2) the *map* operator is applied on each piece of data; (3) the *reduce* operator is used to combine the results; (4) the final result is thus obtained.

To implement the matrix vector multiplication we define: a map operator which compute the product of a matrix and a vector; and a reduce operator which takes i sub-matrices of size (n', m) and assemble them into a $(i \times n', m)$ matrix.

The BSP cost of the BSP algorithm is: $\mathcal{Q}(i) \times T_{map} \oplus \mathcal{Q}(i) \times \mathbf{g} \oplus \mathcal{Q}(i) \times T_{red} \oplus \mathbf{L}$ where $\mathcal{Q}(i)$ stands for the total amount data stored at processor i. The MULTI-BSP cost of the MULTI-BSP algorithm is: $\mathcal{S}(0) \times T_{map} \oplus \sum_{i=1}^{d}(\mathcal{S}(i-1) \times \mathbf{g}_{i-1} \oplus \mathbf{L}_{i-1}) \oplus \mathcal{S}(i) \times T_{red})$ where T_{map} (resp. T_{red}) is the time of the mapping (resp. reducing) and $\mathcal{S}(i)$ stands for the total amount data stored at level i; for example, we have $N \times M/2/2$ elements on each leaf of a dual-core with two thread per core. We assume the following size (quantity of memory) of values such as $SizeOf(\texttt{float}) = 64 Bytes$ and $SizeOf(\texttt{float array}) = n \times SizeOf(\texttt{float})$ if the array contains n elements. We omit small overheads and alternative costs relative to each level for the sake of simplification. Furthermore, the cost of serialisation of the data is taken into account in the $\mathbf{g}$ parameter.

4.2 Algorithms implementation

The BSML codes or mapping/reducing and their descriptions are available in [8,9]. In the context of MULTI-BSP functional programming, we must now write the map/reduce matrix vector product algorithm using the MULTI-ML language. As the MULTI-ML language uses a tree based hierarchical way of executing code, the map/reduces algorithms are almost embedded in the syntax of the language. Indeed, the map phase consists in mapping a function toward the leaves of the MULTI-BSP architecture, while the reduce phase is basically the combination of the results toward the root node.

In the map/reduce implementation, we assume that the values were previously distributed such as each leaves already contains the sub-matrices and nodes are empty. Thus, the distribution is handled by a tree data structure of matrices. As in our implementation a matrix is represented by a one dimension array, the input data is typed α `array` **`tree`**. The map multi-function is written in Fig. 7 (left). As expected, we call recursively the multi-function `map` toward the leaves. When reached, the leaves are going to apply the map operator `f` on their

```
let m_map f tda =
  let multi tree map tda =
     where node =
        let rc = ≪ map tda ≫ in
         finally (rc ,[])
     where leaf =
        f (at tda)
in map tda
```

```
let m_reduce op e tda =
  let multi tree reduce tda =
    where node =
      let rc = ≪ reduce tda ≫ in
      let sub_vals = to_array rc in
      let res = fold_left op e sub_vals in
       finally (rc, res)
    where leaf =
      at tda
in reduce tda;;
```

Fig. 7. Codes of the MULTI-ML mapping (left) and reducing (right).

data stored in tda (the tree distributed array of sub-matrices). Then, we build a tree which contains the results on leaves.

After reaching the leaves using the recursive calls, the reduce multi-function simply retrieve the sub-results of its sub-nodes from rc. It transform the parallel data structure into a local array using to_array and apply the reduce operator of each sub-matrices. Finally, the resulting matrix is used to propagate the result to the root node (Fig. 7, right).

4.3 Performance predictions

Benchmarks were performed on the following architecture: MIREV2 8 nodes, each with 2 quad-cores (AMD 2376 at $2.3Ghz$) with $16GB$ of memory per node and a $1Gbit/s$ network. Based on the computation and communication cost of each phases it is possible to compute the cost of the proposed algorithm. To do so, we use the MULTI-BSP parameters which can be estimated using the probe method [3]. We use the following parameters: $\mathbf{g}_0 = \infty$, $\mathbf{g}_1 = 6$, $\mathbf{g}_2 = 3$ and $\mathbf{g}_0 = 1100$, $\mathbf{g}_1 = 1800$, $\mathbf{g}_2 = 0$ and $\mathbf{L}_0 = 149000$, $\mathbf{L}_1 = 1100$, $\mathbf{L}_2 = 1800$, $\mathbf{L}_3 = 0$. For BSP we get $\mathbf{g} = 1500$ and $\mathbf{L} = 21000$.

$T_{Def} = 2.921$	$T_{Let} = 1.312$	$T_{Get} = 1{,}324$	$T_{BoolAnd} = 0.184$
$T_{Clo} = 0.167$	$T_{Var} = 0.619$	$T_{FloatAdd} = 0{,}881$	$T_{IntEq} = 0.284$
$T_{FunApp} = 1.505$	$T_{Set} = 1{,}778$	$T_{FloatMult} = 1{,}317$	

Table 1. Operator timings in μs.

Thank to a micro-benchmarking library [17] of OCAML, we have estimated the execution time of the main operators which are used in the map operator: multiplication, addition, set and get a value from an array. The timings for each operators are available in Table 1 where T_{mult}, T_{add}, T_{set} and T_{get} are respectively standing for multiplication, addition, affectation and read in an array. We have neglect the times to build the closures (and apply them) for both multi-functions and the recursive functions since most of the computations come from mapping and reducing.

Thus, we have that $T_{map} = 3 \times T_{get} \oplus T_{set} \oplus 2 \times T_{FloatMult} \oplus 3 \times T_{FloatAdd} \oplus 2 \times T_{BoolAnd} \oplus 2 \times T_{IntEq} + 10 \times T_{Var}$ and $T_{red} = T_{get} \oplus T_{set} \oplus 5 \times T_{var} \oplus T_{IntAdd} \oplus T_{IntEq}$. As the cost of such atomic operations are prone to significant variation

because of the compilation optimisation, loops structures and cache mechanisms, we assume that those costs is *"a good approximation"* of the average computation time needed by these operations. A more precise approaches can be found in [12].

The performance prediction compared to the execution time of the matrix vector multiplication can be found in Fig. 8. We perform the tests for both BSML and MULTI-ML. We do not used all the cores since our current MULTI-ML implementation needs specific processes to handle nodes (which is not the case for BSML) and thus we want to be fair for the cost analysis. Note that it is a too small example and BSML is sometime more efficient than MULTI-ML. A comparison between the two languages on bigger examples is available in [2]. The tests has been done for 2 nodes (left) and then for 8 nodes (right).

We can observe that the performance prediction is coherent to the execution time of the algorithm (and its polynomial complexity). The curves slopes are similar even not very accurate. This is mainly due to the fact that the sequential cost of our method is no fine enough. For example, because this is a toy example, we do not use the cache possibilities of the MULTI-BSP model and thus MULTI-ML suffers for some miss-caches that are not currently predicted. The garbage collector of OCAML can also disturb the prediction.

5 Related work

The PRAM [7] family is the oldest way of structuring parallel algorithms. Nevertheless, it is still an accurate way of studying the intrinsic parallelism of algorithmic problems. Close to BSP, the LogP [5] models are, most of the time, used to study network capabilities and low-level libraries such as MPI. Extensions of BSP such as [16,18] were proposed to allows sub-synchronisations. Hierarchical approaches were also proposed in [4]. Parallel algorithmic skeleton are often use to proposed a cost prediction based on a structured approach, as in [11]. An initial attempt was proposed in [10] to compute the BSP cost. In [15], a shape analysis techniques developed in the FISH programming language is used to propose language with an accurate, portable cost model. In the sequential world, Resource Aware ML (RAML) [12] allows to automatically and statically computes the resource-use bounds for OCAML programs. A version for parallel and sequential composition was also proposed in [13].

Those models seems not adapted to our approach as they do not provide both simplicity and accuracy for hierarchical architectures with a structured execution scheme.

6 Conclusion

Overview of the work. In this article we propose a formal semantic with cost annotations allowing cost prediction of MULTI-BSP algorithm. We propose a set of rules adapted to a (core) version of a sequential and purely functional version of ML. Then, we extend this semantics to allows BSP, and then, MULTI-BSP codes. Thanks to this incremental approach, we propose a restrained set of rules allowing a static cost prediction of MULTI-BSP algorithms.

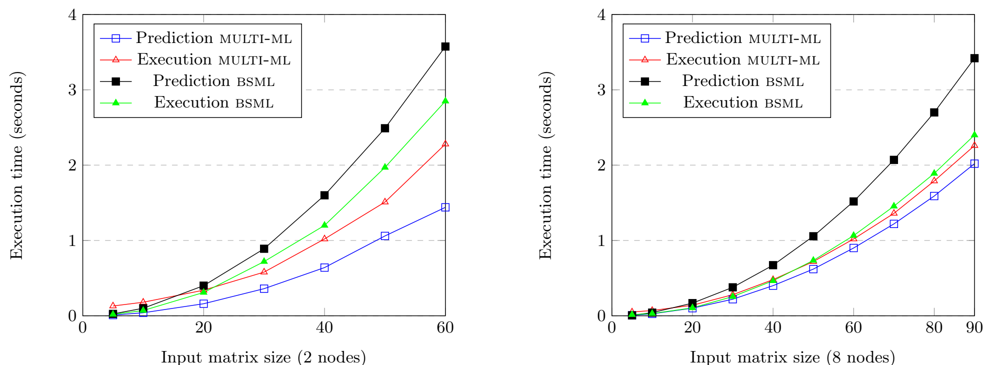

Fig. 8. Performance prediction compared to execution time for BSML and MULTI-ML. For 2 nodes (left) and 8 nodes (right).

To expose the usability of the cost model embedded in the semantics, we compare performance prediction and actual benchmarks on several parallel architectures. As our approach is simplified and consider abstract (BSP and MULTI-BSP) parameters and is based on the estimated execution time of atomic operation, it may suffers from accuracy issues dues to the runtime context (such as cache-misses). We show that our cost estimation is close to the execution time on a simple map/reduce algorithm apply to a matrix-vector multiplication.

Future work. An interesting use of this cost semantic is to propose a analysis able to statically infer a cost of a given algorithm. Such an approach is available for programming imperative BSP algorithm [14] and could be extended to functional MULTI-BSP programming using an approach similar to the one proposed in [12]. Thus it would be possible to give the cost of a program at compile time.

References

1. Allombert, V.: Functional Abstraction for Programming Multi-Level Architectures: Formalisation and Implementation. Ph.D. thesis, UPEC, France (2017)
2. Allombert, V., Gava, F.: Programming BSP and Multi-BSP algorithms in ML. High-Level Parallel Programming and Applications (2018)
3. Bisseling, R.H.: Parallel Scientific Computation: A Structured Approach Using BSP and MPI. Oxford University Press (2004)
4. Cha, H., Lee, D.: H-BSP: A Hierarchical BSP Computation Model. The Journal of Supercomputing **18**(2), 179–200 (Feb 2001)
5. Culler, D., al.: LogP: Towards a Realistic Model of Parallel Computation. In: Principles and Practice of Parallel Programming. pp. 1–12. ACM (1993)
6. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM **51**(1), 107–113 (Jan 2008)
7. Fortune, S., Wyllie, J.: Parallelism in Random Access Machines. In: Theory of Computing. pp. 114–118. ACM (1978)
8. Gava, F.: BSP Functional Programming: Examples of a Cost Based Methodology. In: Computational Science – ICCS 2008, pp. 375–385 (Jun 2008)

9. Gesbert, L., Gava, F., Loulergue, F., Dabrowski, F.: Bulk synchronous parallel ML with exceptions. Future Generation Computer Systems **26**(3), 486–490 (Mar 2010)
10. Hayashi, Y., Cole, M.: BSP-based Cost Analysis of Skeletal Programs. In: Scottish Functional Programming Workshop (SFP99). pp. 20–28 (2000)
11. Hayashi, Y., Cole, M.: Static Performance Prediction of Skeletal Parallel Programs. Parallel Algorithms and Applications **17**(1), 59–84 (Jan 2002)
12. Hoffmann, J., Das, A., Weng, S.C.: Towards Automatic Resource Bound Analysis for OCaml. In: Principles of Programming Languages. POPL 2017, ACM (2017)
13. Hoffmann, J., Shao, Z.: Automatic Static Cost Analysis for Parallel Programs. In: Programming on Programming Languages and Systems - Volume 9032 (2015)
14. Jakobsson, A.: Automatic Cost Analysis for Imperative BSP Programs. International Journal of Parallel Programming (Feb 2018)
15. Jay, C.: Costing parallel programs as a function of shapes. Science of Computer Programming **37**(1), 207–224 (2000)
16. Juurlink, B.H.H., Wijshoff, H.A.G.: The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model. In: Euro-Par'96 (1996)
17. Roshan, J., et al.: Core_bench: Micro-benchmarking library for OCaml (2014)
18. de la Torre, P., Kruskal, C.P.: Submachine locality in the bulk synchronous setting. In: Euro-Par'96 Parallel Processing (1996)
19. Valiant, L.G.: A Bridging Model for Parallel Computation. Commun. ACM **33**(8), 103–111 (Aug 1990)
20. Valiant, L.G.: A Bridging Model for Multi-core Computing. J. Comput. Syst. Sci. **77**(1), 154–166 (Jan 2011)