ARTICLE IN PRESS

Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Future Generation Computer Systems xxx (2004) xxx–xxx

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.com/locate/future

A static analysis for Bulk Synchronous Parallel ML to avoid parallel nesting☆

F. Gava*, F. Loulergue

Laboratory of Algorithms, Complexity and Logic, University Paris Val de Marne, Complexity and Logic, 61, Avenue du Général de Gaulle, 94010 Créteil Cedex, France

Abstract

The BSMLlib library is a library for Bulk Synchronous Parallel (BSP) programming with the functional language Objective Caml. It is based on an extension of the λ-calculus by parallel operations on a data structure named parallel vector, which is given by intention. In order to have an execution that follows the BSP model, and to have a simple cost model, nesting of parallel vectors is not allowed. The novelty of this paper is a type system which prevents such nesting. This system is correct w.r.t. the dynamic semantics.
© 2004 Published by Elsevier B.V.

Keywords: Bulk Synchronous Parallelism; Functional programming; Polymorphic type system

1. Introduction

Bulk Synchronous Parallel ML or BSML is an extension of the ML family of functional programming languages for programming parallel Bulk Synchronous Parallel algorithms as functional programs. Bulk-Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [7] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. BSML expresses BSP algorithms with a small set of primitives taken from the *confluent* BSλ-calculus [3]: a constructor of parallel vectors, asynchronous parallel function application, synchronous global communications and a synchronous global conditional. The BSMLlib library (http://bsmllib.free.fr) implements the BSML primitives using Objective Caml [2] and MPI. It is efficient and its performance follows curves predicted by the BSP cost model.

Our goal is to provide a *certified* programming environment for bulk synchronous parallelism. This environment will contain a byte-code compiler for BSML and a library for the Coq Proof Assistant used to cer-

☆ This paper is a revised version of a paper presented at the Seventh International Conference on Parallel Computing Technologies (PaCT 2003).

* Corresponding author.
E-mail address: gava@univ-paris12.fr (F. Gava); loulergue@univ-paris12.fr (F. Loulergue).

0167-739X/$ – see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.future.2004.05.006

tify BSML programs. One of the advantages of the Objective Caml language (and more generally of the ML family of languages) is its *static polymorphic type* inference. In order to have both simple implementation and cost model that follows the BSP model, nesting of parallel vectors is not allowed. BSMLlib being a library, the programmer is responsible for this absence of nesting. This breaks the *safety* of our environment.

The novelty of this paper is a type system which prevents such nesting. This system is correct w.r.t. the dynamic semantics. We first present the BSP model, give an informal presentation of BSML (Section 2), explain in detail why nesting of parallel vectors must be avoided (Section 3), give our static analysis (Section 4) and conclude (Section 5).

2. Functional Bulk Synchronous Parallelism

A Bulk Synchronous Parallel [6] computer contains a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. Its performance is characterized by 3 parameters expressed as multiples of the local processing speed: the number of processor-memory pairs p, the time l required for a global synchronization and the time g for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation in time gh for any arity h.

A BSP program is executed as a sequence of *super-steps* each one divided into (at most) three successive and logically disjoint phases. In the first phase each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes. In the second phase the network delivers the requested data transfers and in the third phase a global synchronization barrier occurs, making the transferred data available for the next super-step. The execution time of a super-step s is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

$$\mathrm{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} * g + l$$

where $w_i^{(s)}$ = local processing time on processor i during super-step s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during super-step s. The execution time of a BSP program composed of S super-steps is thus $\sum_s \mathrm{Time}(s)$. In general this execution time is a function of p and of the size of data n, or of more complex parameters like data skew and histogram sizes.

There is currently no implementation of a full Bulk Synchronous Parallel ML language but rather a partial implementation as a library for Objective Caml. The so-called BSMLlib library is based on the following elements. It gives access to the BSP parameters of the underling architecture. In particular, it offers the function **bsp_p**: unit → int such that the value of **bsp_p()** is p, the *static* number of processes of the parallel machine. There is also an abstract polymorphic type α **par** which represents the type of p-wide parallel vectors of objects of type α, one per process. The nesting of **par** types is prohibited.

The BSML parallel constructs operate on parallel vectors. Those parallel vectors are created by: **mkpar**: (int → α) → α **par** so that (**mkpar** f) stores (f i) on process i for i between 0 and $(p-1)$. We usually write f as (**fun** pid → e) to show that the expression e may be different on each processor. This expression e is said to be *local*. The expression (**mkpar** f) is a parallel object and it is said to be *global*.

A BSP algorithm is expressed as a combination of asynchronous local computations and phases of global communication with global synchronization. Asynchronous phases are programmed with **mkpar** and with:

apply: (α → β) **par** → α **par** → β **par**

apply (**mkpar** f) (**mkpar** e) stores ((f i)(e i)) on process i. The communication and synchronization phases are expressed by:

put: (int → α option) **par** → (int → α option) **par**

where α option is defined by: **type** α option = None | Some of α.

Consider the expression:

put(**mkpar**(**fun** i → f_i)) (1)

To send a value *v* from process *j* to process *i*, the function f_j at process *j* must be such that $(f_j\ i)$ evaluates to (Some *v*). To send no value from process *j* to process *i*, $(f_j\ i)$ must evaluate to None. Expression (1) evaluates to a parallel vector containing a function g_i of delivered messages on every process *i*. At process *i*, $(g_i\ \ j)$ evaluates to None if process *j* sent no message to process *i* or evaluates to (Some *v*) if process *j* sent the value *v* to the process *i*.

The full language would also contain a synchronous conditional operation, omitted here for the sake of conciseness.

3. Motivations

In this section, we present why we want to avoid nesting of parallel vectors in our language. Let consider the following BSML program:

```
let bcast n vec =
  let tosend = mkpar(fun i v d →
    if i = n then Some v else None) in
  let recv = put(apply tosend vec) in
  apply (replicate noSome)
    (apply recv (replicate n))
```

This program uses the following functions:

```
let replicate x = mkpar(fun pid→ x)
let noSome (Some x) = x
```

(bcast 2 vec) broadcasts the component of the parallel vector *vec* held at process 2 to all other processes. The BSP cost for a call to this program is:

$$p\ + (p-1)\times s\times g\ +\ l \qquad (2)$$

where *s* is the size of the value held at process 2. Consider now the expression:

```
let example1 =
  mkpar(fun pid → bcast pid vec)
```

Its type is (τ **par par**) where τ is the type of the components of the parallel vector *vec*. A first problem is the meaning of this expression. In Section 2, we said that (**mkpar** *f*) evaluates to a parallel vector such that process *i* holds value (*f i*). In the case of this example, it means that process 0 should hold the value of (bcast 0 vec). BSML being based on the confluent calculus [3], it is possible to evaluate (bcast 0 vec) sequentially. But in this case the execution time will not follow the formula (2). The cost of an expression will then depend on its context. The cost model will no more be compositional. We could also choose that process 0 broadcasts the expression (bcast 0 vec) and that all processes evaluate it. In this case the execution time will follow the formula (2). But the broadcast of the expression will need communications and synchronization. This preliminary broadcast is not needed if (bcast 0 vec) is not under a **mkpar**. Thus, we have additional costs that make the cost model still non compositional. Furthermore, this solution would imply the use of a scheduler and would make the cost formulas very difficult to write.

To avoid those problems, nesting of parallel vectors is not allowed. The typing ML programs is well-known [5] but is not suited for our language. Moreover, it is not sufficient to detect nesting of abstract type α **par** such as the previous example. Consider the following program:

```
let example2 =
  mkpar(fun pid →
    let this = mkpar(fun i → i) in pid)
```

Its type is (int **par**) but its evaluation will lead to the evaluation of the parallel vector *this* inside the outmost parallel vector. Thus, we have a nesting of parallel vectors which cannot be seen in the type. Other problems arise with polymorphic values. The most simple example is a projection: **let** fst = **fun** $(a, b) \to a$. Its type is of course $\alpha\times\beta\to\alpha$. The problem is that some instantiations are incorrect. For example:

fst (1, **mkpar**(**fun** $i \to i$))

has type int given by the Objective Caml system. But the evaluation of the expression needs the evaluation of a parallel vector. Thus, we may be in a situation such as in *example2*. One solution would be to have a syntactic distinction between global and local variables (as in the BSλ-calculus). The type system would be simpler but it would be very inconvenient for the programmer since she or he would have to write three different versions of the *fst* function (the fourth possible version is incorrect). The nesting can be even more difficult to detect:

```
let c1 _ = (1, mkpar(fun i → i)
and c2 _ = (2, put(mkpar(fun i d → None))) in
   mkpar(fun i →
      if i < (bsp_p()/2) then fst(c1()) else fst(c2()))
```

The evaluation of this expression would imply the evaluation of $c_1()$ on the first half of the network and $c_2()$ on the second. But **put** implies a synchronization barrier and not **mkpar** so this will lead to mismatched barriers and the behavior of the program will be unpredictable. The goal of our type system is to reject such expressions.

4. A Polymorphic Type System

We begin by defining the term algebra for the basic kinds of semantic objects: the simple types. Simple types are defined by the following grammar:

$$\tau ::= \kappa \mid \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid (\tau\ \mathbf{par})$$

where κ denotes basic types (int, unit, etc.), α variable types, $\tau_1 \to \tau_2$ types of functions, $\tau_1 \times \tau_2$ types of pairs and (τ **par**) types of parallel vectors.

We want to distinguish between three subsets of simple types. The set of local types L, which represent usual Objective Caml types, the variable types V for polymorphic types and global types G, for parallel objects. The local types (written $\dot{\tau}$) are:

$$\dot{\tau} ::= \kappa \mid \dot{\tau}_1 \to \dot{\tau}_2 \mid \breve{\tau} \to \dot{\tau} \mid \dot{\tau}_1 * \dot{\tau}_2$$

the variable types are (written $\breve{\tau}$):

$$\breve{\tau} ::= \alpha \mid \dot{\tau}_1 \to \breve{\tau}_2 \mid \breve{\tau}_1 \to \breve{\tau}_2 \mid \breve{\tau}_1 * \breve{\tau}_2 \mid \breve{\tau}_1 * \dot{\tau}_2 \mid \dot{\tau}_1 * \breve{\tau}_2$$

and the global types (written $\bar{\tau}$) are:

$$\bar{\tau} ::= (\breve{\tau}\, par) \mid (\dot{\tau}\, par) \mid \breve{\tau}_1 \to \bar{\tau}_2 \mid \dot{\tau}_1 \to \bar{\tau}_2 \mid \bar{\tau}_1 \to \bar{\tau}_2$$
$$\mid \bar{\tau}_1 * \bar{\tau}_2 \mid \breve{\tau}_1 * \bar{\tau}_2 \mid \bar{\tau}_1 * \breve{\tau}_2 \mid \dot{\tau}_1 * \bar{\tau}_2 \mid \bar{\tau}_1 * \dot{\tau}_2$$

Of course, we have $L \cap G = \emptyset$ and $V \cap G = \emptyset$. But, it is easy to see that not every instantiations from a variable type are in one of these kinds of types. Take for example, the simple type (α **par**)$\to$ int or the simple type (α **par**) and the instantiation $\alpha =$int **par**: it leads to a nesting of parallel vectors.

To remedy this problem, we will use *constraints* to say which variables are in L or not. For a polymorphic type system, with this kind of constraints, we introduce a *type scheme* with constraints to generically represent the different types of an expression: $\sigma ::= \forall\alpha_1, \ldots, \alpha_n.[\tau/C]$, where τ is a simple type and C is a constraint of classical propositional calculus given by the following grammar:

$$C ::= \mathbf{True} \mid \mathbf{False} \mid \mathcal{L}(\alpha) \mid C_1 \wedge C_2 \mid C_1 \Rightarrow C_2$$

When the set of variables is empty, we simply write $[\tau/C]$ and do not write the constraints when they are equal to **True**. We suppose that we work modulo some usual equations (like associativity) about the $\wedge$ operator.

For a simple type τ, $\mathcal{L}(\tau)$ says that the simple type is in L and we uses the following inductive rules to not have the locality of a type but of its variables:

$$\mathcal{L}(\kappa) = \mathbf{True}$$
$$\mathcal{L}(\tau\ par) = \mathbf{False}$$
$$\mathcal{L}(\tau_1 \to \tau_2) = \mathcal{L}(\tau_1) \wedge \mathcal{L}(\tau_2)$$
$$\mathcal{L}(\tau_1 * \tau_2) = \mathcal{L}(\tau_1) \wedge \mathcal{L}(\tau_2)$$

In the type system and for the substitution of a type scheme we will use rules to construct constraints from a simple type called *basic constraints*. We note C_τ for the basic constraints from the simple type τ and we use the following inductive rules:

$$C_\tau = \mathbf{True} \quad \text{if}\ \tau\ \text{atomic}$$
$$C_{(\tau_1 \to \tau_2)} = C_{\tau_1} \wedge C_{\tau_2} \wedge \mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1)$$
$$C_{(\tau\ par)} = \mathcal{L}(\tau) \wedge C_\tau$$
$$C_{(\tau_1 * \tau_2)} = C_{\tau_1} \wedge C_{\tau_2}$$

In our type system, we will use basic constraints and constraints associated to sub-expressions that are needed in cases similar to the expression *example2* of the previous section.

The set of free variables of a type scheme is defined by:

$$\mathcal{F}(\forall\alpha_1,\ldots,\alpha_n.[\tau/\mathcal{C}]) = (\mathcal{F}(\tau)\cup\mathcal{F}(C))\setminus\{\alpha_1,\ldots,\alpha_n\}$$

where the free variables of the type and the constraints are defined by trivial structural induction. We note Dom for the domain of a *substitution* (i.e. a finite application of variables of type to simple types). We can now define the substitution on a type scheme.

Definition 1. If $\alpha_1,\ldots,\alpha_n$ are out of reach of φ,

$$\varphi(\forall\alpha_1,\ldots,\alpha_n.[\tau/C]) = \forall\alpha_1,\ldots,\alpha_n. \quad [\varphi(\tau)/\varphi(C) \bigwedge_{\beta_i\in\mathrm{Dom}(\varphi)\cap\mathcal{F}([\tau/C])} C_{\varphi(\beta_i)}] \tag{3}$$

We say that a variable α is *out of reach* of a substitution φ if: $\varphi(\alpha)=\alpha$, i.e. φ don't modify α (or α is not in the domain of φ) and if α is not free in $[\tau/C]$, then α is not free in $\varphi([\tau/C])$, i.e., φ do not introduce α in its result. The condition that $\alpha_1,\ldots,\alpha_n$ are out of reach of φ can always be validated by renaming first $\alpha_1,\ldots,\alpha_n$ with fresh variables (we suppose that we have an infinite set of variables). The substitution on the simple type and on the constraints are defined by trivial structural induction.

A type scheme can be seen like the set of types given by instantiation of the quantifier variables. We introduce the notion of instance of a type scheme with constraints.

Definition 2. We note $[\tau/C] \leq \forall\alpha_1,\ldots,\alpha_n.[\tau'/C']$ iff there exists a substitution φ of domain $\alpha_1,\ldots,\alpha_n$ where $[\tau/C]=\varphi([\tau'/C'])$.

We write E for an *environment* which associates type schemes to free variables of an expression. It is an application from free variables (identifiers) of expressions to type schemes. We note $\mathrm{Dom}(E)=\{x_1,\ldots,x_n\}$ for its domain, i.e. the set of variables associated. We assume that all the identifiers are distinct. We note $E(x)$ for the type scheme associated with x in E. The substitution φ on E is a point to point substitution on the domain of E. The set of free variables is naturally defined on the free variables on all the type scheme associated in the domain of E. Finally, we write $E+\{x:\sigma\}$ for the extension of E to the mapping of x to σ. If, before this operation, we have $x\in\mathrm{Dom}(E)$, we can replace the range by the new type scheme for x. To continue with the introduction of the type system, we define how to generalize a type scheme. Yet, type schemes have universal quantified variables, but not all the variables of a type scheme can.

Definition 3. Given an environment E, a type scheme $[\tau/C]$ without universal quantification, we define an operator Gen to introduce universal quantification: $Gen([\tau/C],E)=\forall\alpha_1,\ldots,\alpha_n.[\tau/C]$, where$\{\alpha_1,\ldots,\alpha_n\}=\mathcal{F}(\tau)\setminus\mathcal{F}(E)$

With this definition, we have introduced polymorphism. The universal quantification gives the choice for the system to take the good type from a type scheme.

We note *TC* the function which associates a type scheme to the constants and to the primitive operations (some cases):

$$\begin{aligned}
&TC(i) = int \quad i=0,1,\ldots\\
&TC(\mathbf{fst}) = \forall\alpha\beta.[(\alpha*\beta)\to\alpha/\mathcal{L}(\alpha)\Rightarrow\mathcal{L}(\beta)]\\
&TC(\mathbf{mkpar}) = \forall\alpha.[(int\to\alpha)\to(\alpha\ par)/\mathcal{L}(\alpha)]\\
&TC(\mathbf{apply}) = \forall\alpha\beta.[((\alpha\to\beta)par*(\alpha\ par))\\
&\qquad\qquad\to(\beta\ par)/\mathcal{L}(\alpha)\wedge\mathcal{L}(\beta)]\\
&TC(\mathbf{put}) = \forall\alpha.[(int\to\alpha)par\\
&\qquad\qquad\to(int\to\alpha)par/\mathcal{L}(\alpha)]
\end{aligned}$$

We formulate type inference by a deductive proof system that assigns a type to an expression of the language. The context in which an expression is associated with a type is represented by an environment which maps identifiers to type schemes. Deductions produce conclusions of the form $E\vdash e:[\tau/C]$ which are called *typing judgments*, they could be read as: "in the type environment E, the expression e has the type $[\tau/C]$". The static semantics manipulates type schemes by using the mechanism of generalization and instantiation specified in the previous sections. The inductive rules of the type system are given in the Fig. 1.

In all the inductive rules, if a constraint C of the hypothesis can be solved to **False** then the inductive rule cannot be applied and then the expression is not well typed. Our constraints are a sub part of the propositional calculus, so solving constraints is decidable.

$$\frac{[\tau/C] \leq E(x)}{E \vdash x : [\tau/C]}(\text{Var}) \qquad \frac{[\tau/C] \leq TC(c)}{E \vdash c : [\tau/C]}(\text{Const}) \qquad \frac{[\tau/C] \leq TC(op)}{E \vdash op : [\tau/C]}(\text{Op})$$

$$\frac{E + \{x : [\tau_1/C_{\tau_1}]\} \vdash e : [\tau_2/C_2]}{E \vdash (\textbf{fun}\ x \to e) : [\tau_1 \to \tau_2/C_{(\tau_1 \to \tau_2)} \wedge C_2]}(\text{Fun})$$

$$\frac{E \vdash e_1 : [\tau' \to \tau/C_1] \qquad E \vdash e_2 : [\tau'/C_2]}{E \vdash (e_1\ e_2) : [\tau/C_1 \wedge C_2]}(\text{App})$$

$$\frac{E \vdash e_1 : [\tau_1/C_1] \qquad E + \{x : Gen([\tau_1/C_1], E)\} \vdash e_2 : [\tau_2/C_2]}{E \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : [\tau_2/C_1 \wedge C_2 \wedge \mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1)]}(\text{Let})$$

$$\frac{E \vdash e_1 : [\tau_1/C_1] \qquad E \vdash e_2 : [\tau_2/C_2]}{E \vdash (e_1, e_2) : [\tau_1 * \tau_2/C_1 \wedge C_2]}(\text{Pair})$$

$$\frac{E \vdash e_1 : [\textbf{bool}/C_{e_1}] \quad E \vdash e_2 : [\tau/C_{e_2}] \quad E \vdash e_3 : [\tau/C_{e_3}]}{E \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : [\tau/C_{e_1} \wedge C_{e_2} \wedge C_{e_3}]}(\text{Ifthenelse})$$

Fig. 1. The inductive rules.

Like traditional static type systems, the case (*Op*), (*Const*) and (*Var*) used the definition of an instance of a type scheme. The rule (*Fun*) introduces a new type scheme in the environment with the basic constraints from the simple type of the argument. In the rules (*App*), (*Pair*) and (*Let*) we make the conjunction of the constraints to known if the two sub-cases are coherent. Moreover, in (*Let*), we introduce the constraint $\mathcal{L}(\tau_2) \Rightarrow \mathcal{L}(\tau_1)$ because an expression like **let** $x = e_1$ **in** e_2 can be seen like ((**fun** $x \to e_2$) e_1). So we have to protect our type system against functional expression from global values to local values like in the program *example2*. The rule (*ifthenelse*) makes also the conjunction of the constraints. The **if then else** construction could return a global or a usual value.

The basic constraints are important in our type system but are not sufficient. For example, for a parallel identity:

fun $x \to$ **let** y = **put**(**mkpar**(**fun** $i\ d \to$ None)) **in** x

the simple type given by Objective Caml is $\alpha \to \alpha$ and the basic constraints ($\mathcal{L}(\alpha) \Rightarrow \mathcal{L}(\alpha)$) are always solved to **True**. But it is easy to see that the variable x (of type α) could not be a usual value. Our type system, with constraints from the sub-expression (here **put**) would give the type scheme: $[\alpha \to \alpha/\mathcal{L}(\alpha) \Rightarrow$ **False**] (i.e., α could not be a usual value and the instantiation are in *G*). [1] presents the dynamic semantics of BSML and elements of the proof of correctness of the type system with respect to this semantics.

5. Conclusions and future work

BSML allows direct mode Bulk Synchronous Parallel (BSP) programming. To preserve a compositional cost model derived form the BSP cost model, the nesting of parallel vectors is forbidden. The type system presented in this paper allows a static avoidance of nesting. Thus, the pure functional subset of BSML is safe. We have also designed an algorithm for type inference and implemented it.

A general framework for type inference with constrained types [4] also exists and could be used for a type system with only basic constraints. We do not used this system mainly because it has been proved for λ-calculus and not for the BSλ-calculus with its two level structure (local and global), and also because in this system, there are no constraints depending on sub-expressions.

A further work will concern imperative features. Dynamic semantics of the interaction of imperative features with parallel operations have been designed. To ensure statically the safety of this interaction, we are currently working on the typing of effects.

Acknowledgements

The authors wish to thank the anonymous referees for their comments. This work is supported by the ACI Grid program (project CARAML, www.caraml.org) and by a research grant from the French Ministry of Research. French Foreign Office provided a travel grant.

References

[1] F. Gava, F. Loulergue, A polymorphic type system for Bulk Synchronous Parallel ML, in: V. Malyshkin (Ed.), PaCT 2003, No. 2763 in LNCS, Springer, 2003, pp. 215–229.

[2] X. Leroy, The Objective Caml System 3.07, 2003. http://www.ocaml.org.

[3] F. Loulergue, G. Hains, C. Foisy, A calculus of functional BSP programs, Sci. Comp. Progr. 37 (1–3) (2000) 253–277.

[4] M. Odersky, M. Sulzmann, M. Wehr, Type inference with constrained types, Theory Pract. Object Syst. 5 (1) (1999) 35–55.

[5] R. Milner, A theory of type polymorphism in programming, J. Comp. Syst. Sci. 17 (3) (1978) 348–375.

[6] D.B. Skillicorn, J.M.D. Hill, W.F. McColl, Questions and answers about BSP, Scientific Progr. 6 (3) (1997) 249–274.

[7] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103.

Frederic Gava obtained his MS Degree in Computer Science from the University Paris 7 in 2002. He is currently a PhD Student in the University of Paris Val de Marne. His research interests are parallel and grid computing systems, particularly their design and proofs of their properties.

Frederic Loulergue obtained his PhD Degree in Computer Science from the University Orleans in 2000. He is currently an associate professor in the University of Paris Val de Marne. His research interests are high level languages for parallel and grid computing systems.