A Functional Language for Departmental Metacomputing

Gava, F. and Loulergue, F.

Technical Report TR-2004-09

Laboratory of Algorithms, Complexity and Logic
University of Paris XII, Val-de-Marne
61, avenue du Général de Gaulle
F-94010 CRÉTEIL Cedex – France
Tel: +33 (0)1 45 17 16 47
Fax: +33 (0)1 45 17 66 01

A Functional Language for Departmental Metacomputing

Frédéric Gava and Frédéric Loulergue
Laboratory of Algorithms, Complexity and Logic
University Paris Val-de-Marne
61, avenue du Général de Gaulle
94010 Créteil cedex – France
{gava,loulergue}@univ-paris12.fr

TR-2004-08

Abstract

We have designed a functional data-parallel language called BSML for programming bulk synchronous parallel (BSP) algorithms. Deadlocks and indeterminism are avoided and the execution time can be then estimated. For very large scale applications more than one parallel machine could be needed. One speaks about metacomputing. A major problem in programming application for such architectures is their hierarchical network structures: latency and bandwidth of the network between (parallel) nodes could be orders of magnitude worse than those inside a parallel node. Here we consider how to extend both the BSP model and BSML, well-suited for parallel computing, in order to obtain a model and a functional language suitable for metacomputing.

1 Introduction

Some problems require performance that can only be provided by massively parallel computers. For very large scale applications more than one parallel machine could be needed. One speaks about metacomputing. In recent years there has been a trend towards using a set of parallel machine for these kinds of problems. Metacomputing infrastructures couple multiple clusters or parallel machines via a wide-area network. Research on global computational infrastructures has raised considerable interest in running parallel applications on distributed systems. Programming this kind of *metacomputers* is still difficult due to the presence of different networks (local and wide-area). High-level language and formal cost models are needed to ease the programming of these *hierarchical* architectures, a "cluster of clusters".

Bulk-Synchronous Parallel ML or *BSML* is an extension of ML for programming *direct-mode* Bulk Synchronous Parallel algorithms as functional programs. Bulk-Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [29] 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 them with a small set of primitives taken from the *confluent* BSλ calculus [19].

But metacomputing programs need a more detailed model, including latency and bandwidth of the local and wide-area (or intranet) networks, the number of clusters or parallel machines and the number of processors in each parallel machine. We currently make some simplifying assumptions about the networks: we use stable topologies, latencies and bandwidth. When the parallel machines

are still in the same organization, university or building, this kind of programming is usually called *departmental metacomputing*. In this way, regular network performances are certainly realistic for the duration of a part of a program and are still less sensitive to security measures. Assuming a metacomputer with stable network performances allows us to focus on the impact of network performance to design a cost model for this kind of architecture and a functional language for a high-level programming point of view. Our ultimate goal is to develop a functional language which could go beyond this limitations.

This paper describes a further step after [12] towards this direction. In section 2 we briefly review the BSP model and how to extend it for departmental metacomputing by adding a new level of communication. Then, we present informally our new functional parallel language, called DMML, (section 3), a formal semantics (section 4) for a core sub-language and a formal cost model for it (section 5). Section 7 is devoted to an example of collective communication operation. We discuss related work (section 8) and conclude (section 9).

2 A Model for Departmental Metacomputing

We assume throughout this paper that a "metacomputer" is a set of multiple clusters or parallel machines with fully connected local networks (LAN) and a fully connected intranet network (here excessively called WAN or "departmental WAN"). Each parallel machine has a gateway that connects its private LAN to the departmental WAN. [21] gives a classification of "parallel machines" among GRID, Meta or Global computers and the considered structure is usually called *cluster of clusters*. Here we give the name of *departmental metacomputing* system because the parallel machines are within the same organization. In this way, we use regular topologies, constant latencies and bandwidth. A "metacomputer" being a set of clusters or parallel machines, we can use the BSP model for each node and a model for the coordination of the set of parallel nodes.

2.1 Bulk Synchronous Parallelism

A BSP computer contains a set of uniform *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* (for the sake of conciseness, we refer to [28] for more details). In this model, a parallel computation is divided in *super-steps*, at the end of which a barrier synchronization and a routing is performed. Hereafter all requests for data which have been posted during a preceding super-step are fulfilled.

The performance of the machine is characterized by 3 parameters expressed as multiples of the local processing speed s: p is the number of processor-memory pairs, l is the time required for a global synchronization and g is the time 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 $g \times h$ for any arity h. The execution time of a super-step is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time.

2.2 Discussion about the BSP model

There are two main arguments against using BSP for metacomputing. First the global synchronization barrier is claimed to be expensive especially for a set of parallel machines. Second, this model does no take into account the different capacities of the parallel machines and different networks: it is not a heterogeneous and hierarchical model of computation. Our proposal attempts to give a

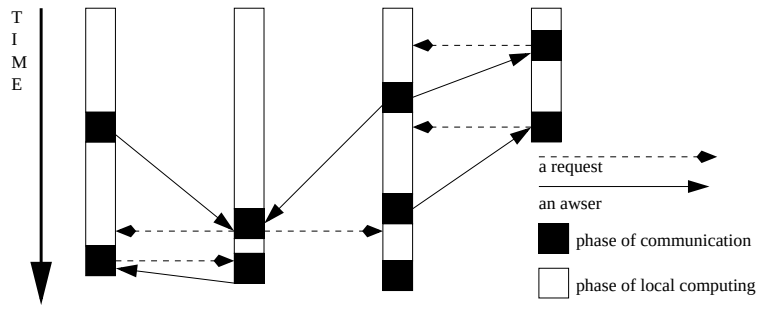

Figure 1: The MPM model of computation

solution to the aforementioned problems. Starting from BSP, we address the problem of enlarging the number of parameters without introducing an unbearable complexity. The result model of computation called DMM is introduced in this section.

To remedy the two first problems, [26] introduces the MPM model of computation which is a model directly inspired by the BSP model. It offers to replace the notion of super-step by the notion of m-step defined as: at each m-step, each process performs a sequential computation phase, then a communication phase (Figure 1, colored boxes are phase of communication). During this communication phase, the processes exchange the data they need for the next m-step.

To remedy the second problem, [20] investigated a two-level hierarchical BSP Model for a cluster of SMP machines without subset synchronization and two-levels of communications (contrary to [9] which have the same parameters as the BSP model and allows subset synchronization). A BSP^2 computer consists of a number of *uniformly* BSP units, connected by a communication network. Execution of a BSP^2 program proceeds in *hyper-steps* separated by global synchronizations. On each hyper-step each BSP unit performs a complete BSP computation (some super-steps) and communicates some data with other BSP units. However, the authors noted that none of the algorithms they have analyzed shows any significant benefit from this approach and the experiments do not follow the model. The failure of the BSP^2 model to provide any major performance comes from three main reasons: first the BSP units are generally different in practice, second the synchronization time for all the BSP units is too expensive and third, the only algorithms that have been considered are classical BSP algorithms without any irregularity or data locality problems.

2.3 A Departmental Metacomputing Cost Model

The BSP^2 model introduces an interesting idea: using the BSP model on each parallel computer and using an additional cost model for metacomputing. To reuse the work done on BSP algorithms and to deal with the different architectures of each parallel computer and with the asynchronous nature of some programs, we propose a two-tiered model. Using the BSP model on each parallel units and the MPM model for coordinating these heterogeneous set of BSP units (to have asynchronous computations).

Thus, a metacomputer in this model is characterized by the following parameters: P the number of parallel computers, L the latency of the WAN, G the time needed to exchange one word between two units, $\mathcal{P} = \{p_0, \ldots, p_{P-1}\}$ the list of the number of processes for each BSP unit (p_j for $0 \leq j < P-1$ the number of processes of the j^{nth} BSP units). In the same way $\mathcal{L} = \{l_0, \ldots, l_{P-1}\}$ is the list of the times needed for one synchronization barrier in each of the BSP units; $\mathcal{S} = \{s_0, \ldots, s_{P-1}\}$ the processors speed of each BSP unit; $\mathcal{G} = \{g_0, \ldots, g_{P-1}\}$ the time for collectively delivering a 1-relation on each BSP unit. We recall that any network of a BSP unit j can deliver an h-relation

in time $g_j \times h$ for any arity h.

We suggest replacing the notion of hyper-step by the notion of d-step (for departmental step) defined as: at each d-step, each BSP unit performs a parallel computation phase as a sequence of super-steps then a communication phase to exchange values between BSP units.

Usually a message sent from a processor within a parallel unit to another parallel unit first goes to the gateway (local network), travels trough the wide-area network to the destination gateway and is then routed to the destination processor through the local network of the destination unit.

During this communication phase the processes exchange the needed data for the next d-step. The model uses the set $\Omega_{d,i}$ for a BSP unit i and a d-step d (BSP units included in $\Omega_{s,i}$ are called "incoming partners") defined as:

$$\Omega_{d,i} = \{j/\text{BSP unit } i \text{ received some messages from the BSP unit } j \text{ at d-step } d\}$$

and $\Phi_{d,i}$ which is inductively defined as:

$$\begin{cases} \Phi_{1,i} = & \max\limits_{j\in\Omega_{1,i}}(W_{(1,j)}, W_{(1,i)}) + \sum\limits_{j\in\Omega_{1,i}}((g_i + G + g_j) \times h^j_{1,i} + l_j) + (g_i \times h^1_i + l_i) + L) \\ \Phi_{d,i} = & \max\limits_{j\in\Omega_{d,i}}(\Phi_{d-1,j} + W_{(d,j)}, \Phi_{d-1,i} + W_{(d,i)}) + \sum\limits_{j\in\Omega_{d,i}}((g_i + G + g_j) \times h^j_{d,i} + l_j) \\ & \qquad\qquad +(g_i \times h^d_i + l_i) + L) \end{cases}$$

where $h^j_{d,i}$ denotes the number of words received by the BSP unit i from the BSP unit j ("incoming partner") during the d-step d for $i \in \{0, \ldots, P-1\}$ and h^d_i is the maximum number of words communicated by a processor of the BSP unit i (incoming or outgoing bytes) during the d-step d and $W_{(d,j)}$ is the sum of the super-steps of a BSP units j:

$$W_{(d,j)} = \sum_{a=0}^{s^j_d}(\max_{b=0}^{p_j}(w^a_b)) \ + \ (\sum_{a=0}^{s^j_d}(\max_{b=0}^{p_j}(h^{(a)}_b))) \times g_j \ + \ (s^j_d \times l_j)$$

where s^j_d is the number of super-steps of the BSP unit j during the d-step d and w^a_b the local processing time on processor b during each super-step a and $h^{(a)}_b = \max\{h^{(a)}_{b+}, h^{(a)}_{b-}\}$ where $h^{(a)}_{b+}$ (resp. $h^{(a)}_{b-}$) is the number of words transmitted (resp. received) by processor b during each super-step a.

The execution time for a complete program is thus bounded by:

$$\Psi = \max\{\Phi_{R,j}/j \in \{0, 1, \ldots, P-1\}\}$$

where R is the number of d-steps of the program. The DMM model takes into account that a BSP unit only synchronizes with each of its incoming partner and is therefore more accurate than the BSP^2 model. More algorithms for irregular problems could be analyzed efficiently.

3 The DMML Language

There is currently no implementation of a full DMML language but rather a partial implementation as a library for Objective Caml language [17, 10, 25]. In this section we describe the core of the BSMLlib library and its extension for programming DMM algorithms.

3.1 The BSMLlib library

The core of so-called BSMLlib is based on the elements given in Figure 2. It gives access to the BSP parameters of the underling architecture. In particular, **bsp_p**() is p, the *static* number of processes. There is 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. Our type system enforces this restriction [13].

The BSML parallel constructs operates on parallel vectors. Those parallel vectors are created by **mkpar** so that (**mkpar** f) stores (f i) on process i for i between 0 and $(p-1)$. We usually write f as **fun** pid $\rightarrow$ 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*.

bsp_p: unit $\rightarrow$ int
mkpar: (int $\rightarrow$ α) $\rightarrow$ α **par**
apply: (α $\rightarrow$ β) **par** $\rightarrow$ α **par** $\rightarrow$ β **par**
type α option = None | Some **of** α
put: (int $\rightarrow$ α option) **par** $\rightarrow$ (int $\rightarrow$ α option) **par**
at: α **par** $\rightarrow$ int $\rightarrow$ α

Figure 2: BSML Core

A BSP algorithm is expressed as a combination of asynchronous local computations (first phase of a super-step) and phases of global communication (second phase of a super-step) with global synchronization (third phase of a super-step). Asynchronous phases are programmed with **mkpar** and **apply**. The expression (**apply** (**mkpar** f) (**mkpar** e)) stores ((f i)(e i)) on process i.

Readers familiar with BSPlib [14] will observe that we ignore the distinction between a communication request and its realization at the barrier. The communication and synchronization phases are expressed by **put**. Consider the expression:

$$\textbf{put}(\textbf{mkpar}(\textbf{fun}\ \text{i} \rightarrow \text{fs}_i)) \tag{1}$$

To send a value v from process j to process i, the function fs_j at process j must be such that (fs_j `i`) evaluates to Some v. To send no value from process j to process i, ($\texttt{fs}_j$ `i`) must evaluate to None. Expression (1) evaluates to a parallel vector containing a function fd_i of delivered messages on every process. At process i, (fd_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 projection operation **at** where (**at** vec n) return the n^{th} value of the parallel vector vec. **at** expresses communication and synchronization phases. Without it, the global control cannot take into account data computed locally. Global conditional is necessary of express algorithms like:

Repeat Parallel Iteration **Until** Max of local errors $< \epsilon$

The projection should not be evaluated inside the scope of a **mkpar**. This is enforced by our type system [13].

The following program is a small example of a direct broadcast algorithm in BSML, where noSome is such as noSome(Some x)=x:

exception Bcast
let replicate x = **mkpar**(**fun** pid $\rightarrow$ x)

```
let parfun f vv = apply (replicate f) vv
let bcast_direct root vv =
  if root<0||root>=bsp_p() then raise Bcast else
  let mkmsg = mkpar(fun pid v dst → if pid=root then Some v else None) in
  parfun noSome (apply (put (apply mkmsg vv)) (replicate root))
```

3.2 The DMMLlib library

The DMMLlib extends the BSMLlib by adding new operators on a new level called departmental. The core of this library is given in Figure 3. It offers functions to access to the parameters of the metacomputer, in particular, the function **dm_p**:unit → int (resp. **dm_g** and **dm_l**) is such that the value of **dm_p**() is P, the static number of BSP units (resp. G and L time of communication and latency of the "departmental WAN"). Parameters of the BSP units are available through the functions **dm_bsp_p**, **dm_bsp_s**, **dm_bsp_g** and **dm_bsp_l**. For example (**dm_bsp_p** a) gives the number of processors of the a^{th} parallel machine.

dm_bsp_p	:	int → int	**dm_p**: unit → int
dm_bsp_s	:	int → float	**dm_g**: unit → float
dm_bsp_g	:	int → float	**dm_l**: unit → float
dm_bsp_l	:	int → float	
mkdep	:	(int → α) → α **dep**	
applydep	:	(α → β) **dep** → α **dep** → β **dep**	
get	:	(int → int → int option) **par dep** → (int → α option) **par dep** → (int → int → α option) **par dep**	
atdep	:	α **par dep** → int → int → α	

Figure 3: DMMLlib Core

There is also a new polymorphic type α **dep** which represents the type of P-wide departmental vectors of objects of type α one per BSP unit. The nesting of **dep** into **dep** or into **par** types is prohibited. But the α of a **dep** type could be either a usual Objective Caml value or a parallel BSMLlib value.

The DMML departmental constructs operates on departmental vectors. Those vectors are created by **mkdep** so that (**mkdep** f) stores (f a) on the BSP unit a for a between 0 and $(P-1)$. The BSML parallel values should not be evaluated outside the scope of a **mkdep**. This could be enforced by a type system, but for the moment, the programmer is responsible for respecting this rule. The BSML parallel constructs operate on parallel vectors of size p_a for the BSP unit a. For example in the scope of a **mkdep**, (**mkpar** f) stores (f i) on process i for i between 0 and $(p_a - 1)$ for the BSP unit a.

A DMM algorithm is expressed as a combination of asynchronous BSP computations (first phase of a d-step) and phases of communication (second phase of a d-step). Asynchronous phases are programmed with **mkdep** and with **applydep**. This operations is such as on BSP unit a, (**applydep** (**mkdep** f) (**mkdep** e)) stores ((f a)(e a))

The communication phases of a d-step are expressed by **get**. Consider the expression:

$$\begin{array}{ll} \textbf{get} & (\textbf{mkdep}(\textbf{fun}\ \text{a} \rightarrow \textbf{mkpar}(\textbf{fun}\ \text{i} \rightarrow \text{f}_{a,i}))) \\ & (\textbf{mkdep}(\textbf{fun}\ \text{b} \rightarrow \textbf{mkpar}(\textbf{fun}\ \text{j} \rightarrow \text{v}_{b,j}))) \end{array} \quad (2)$$

For a process i of the BSP unit a, to receive the n^{th} value from the process j of the BSP unit

b (it is an incoming partners), the function $f_{a,i}$ at process i of the BSP unit a must be such that ($f_{a,i}$ b j) evaluates to Some n. To receive no value ($f_{a,i}$ b j) must evaluate to None.

Our expression evaluates to a departmental vector containing parallel vectors of functions $f'_{a,i}$ of delivered messages on every process of every BSP unit.

At process i of the BSP unit a ($f'_{a,i}$ b j) evaluates to None if process i of the BSP unit a receives no message from process j of the BSP unit b or if ($v_{b,j}$ n) evaluates to None (the process j of the BSP unit b does not have a n^{th} value and sends the empty value). It also evaluates to Some $v^n_{b,j}$ if it received a value from the process j of the BSP unit b and if ($v_{b,j}$ n) evaluates to (Some $v^n_{b,j}$).

There is also a projection operation **atdep**. It is used in the same way as the **at** operation but it takes as arguments a departmental vector of parallel vectors and two integers being the number of the cluster and the number of the process considered. This operation should not be evaluated inside a **mkdep**. Using **atdep** the global behavior of the program could depend on a local value.

The following program is a small example of a direct broadcast algorithm in DMML:

```
let replicate_all x = mkdep (fun a  → replicate x)
let apply_all gf gv = applydep(applydep(mkdep(fun a → fun f → fun v → apply f v)) gf) gv
let parfun_all f x = apply_all (replicate_all f) x

let get_one_all datas srcs =
  let to_send= parfun_all(fun v n  → Some v) datas
  and srcs_mod = parfun_all (fun (a,i)  → let ap = (natmod a (dm_p())) in
  (ap,(natmod i (dm_bsp_p ap)))) srcs in
  let ask = parfun_all (fun (a,i)  → fun cluster pid  →
    if (cluster=a)&&(pid=i) then Some 0 else None) srcs_mod in
  parfun2_all (fun f → fun (a,i) → (noSome (f a i))) (get ask to_send) srcs_mod

let bcast_direct_all rcluster rootpid vv =
  if rcluster<0||rootpid<0||rcluster>=dm_p()||roopid>=(dm_bsp_p rcluster)
  then raise Bcast
  else get_one_all vv (replicate_all(rcluster,rootpid))
```

Section 7 presents a less naive DMM version of broadcast.

The operators presents informally is this section are taken from a core language which formal semantics in given in the next section. This semantics is confluent. So the determinism of DMML programs is guaranteed. As in usual functional languages, we could *prove* the correctness of the DMML implementations of DMM algorithms with a proof assistant as done in [11] for BSP algorithms implemented in BSML. Using the *extraction* capability, we could generate a *certified* implementation to be used either with the sequential or with the parallel implementation of the DMMLlib library.

3.3 Advantages of Functional Deparmental Metacomputing

The clarity, abstraction and formal semantics of functional language make them desirable vehicles for complex software. The functional approach of this parallel model allows the re-use of suitable technical from functional languages because a few numbers of parallel operators is needed. Those operators (for a static number of parallel machines and processes) of a functional language (derived from a confluent core-language see Section 4) are transparent in the sense of making complexity explicit in the syntax. So parallel algorithms programming with DMML are also confluent and keep the advantages of our model.

One important benefit of cost models (as BSP, MPM and DMM) is the ability to accurately predict the execution time requirements of parallel algorithms (communications are clearly separated

from synchronization and computations). The super-steps and d-steps also separate communication and local calculus which avoid deadlocks.

Also, as in functional languages, we could *prove* and *certify* functional implementation of those algorithms with a proof assistant as in [11] and using its *extraction* possibility, generate a *certified* implementation to be used independently of the sequential or parallel implementation of the DMML operators.

4 Formal Semantics

Reasoning on the complete definition of a functional and parallel language such as DMML, would have been complex and tedious. In order to simplify the presentation and to ease the formal reasoning, this section introduces a core language and its "small-steps semantics".

4.1 The Mini-DMML core-language

The expressions of mini-DMML (programming syntax), written as e possibly with a prime or subscript, have the following abstract syntax:

e	::=	x	variables	c	constants
		op	operators	**fun** $x \rightarrow e$	abstraction
		$(e\ e)$	application	**let** $x = e$ **in** e	binding
		(e, e)	pairs	**if** e **then** e **else** e	conditional
		mkpar e	parallel vector	**apply** $e\ e$	parallel application
		put e	parallel communication	**at** $e\ e$	projection
		mkdep e	departmental vector	**applydep** $e\ e$	departmental application
		get $e\ e$	departmental communication	**let rec** $g\ x = e$ **in** e	recursive local binding
		atdep $e\ e\ e$	projection		

In this grammar, x or g ranges over a countable set of identifiers. The form $(e\ e')$ stands for the application of a function or an operator e, to an argument e'. The form **fun** $x \rightarrow e$ is the so-called and well-known lambda-abstraction that defines the first-class function whose parameter is x and whose result is the value of e. Constants c are () (the only value of type unit in Objective Caml), the integers, the booleans and the value **nc** (which stands for no communication) which plays the role of the `None` constructor in Objective Caml for the **put** communication operation. The set of primitive operations op contains arithmetic and boolean operations, **fst** and **snd** operators, the whole parameters of the machine and the test function **isnc** of the **nc** constant. We also have our parallel and departmental constructors and the recursive local binding is used to define natural iteration functions (to have more expressiveness).

Before presenting the dynamic semantics of our core-language, i.e., how the expressions of mini-DMML are computed to *values*, we present the values themselves. There is one semantics per value of the machine parameters (number of parallel machine and of processes). In the following, the expressions are now extended with enumerated vectors: parallel vectors $\langle e, \ldots, e \rangle$ and departmental vectors $\phi\!\langle e, \ldots, e \rangle\!\phi$. P denotes the number of parallel machines and $\forall j \in \{0, \ldots, P-1\}$, p_j is the number of processors of the parallel machine j. The values of mini-DMML are defined by the following grammar:

v	::=	**fun** $x\ \rightarrow e$	functional value	c	constant
		op	primitive	(v, v)	pair value
		$\langle v, \ldots, v \rangle$	p-wide parallel vector value	$\phi\!\langle v, \ldots, v \rangle\!\phi$	P-wide departmental value

$$
\begin{array}{rcl}
(\textbf{bsp\_p}\ ()) & \xrightarrow[\bowtie_j]{\varepsilon} & p_j \\
\textbf{mkpar}\ v & \xrightarrow[\bowtie_j]{\varepsilon} & \langle (v\ 0), \ldots, (v\ (p_j-1)) \rangle \\
\textbf{apply}\ \langle v_0, \ldots, v_{p_j-1} \rangle\ \langle v'_0, \ldots, v'_{p_j-1} \rangle & \xrightarrow[\bowtie_j]{\varepsilon} & \langle (v_0\ v'_0), \ldots, (v_{p_j-1}\ v'_{p_j-1}) \rangle \\
\textbf{at}\ \langle \ldots, v_n, \ldots \rangle\ n & \xrightarrow[\bowtie_j]{\varepsilon} & v_n \\
\textbf{put}\ \langle v_0, \ldots, v_{p_j-1} \rangle & \xrightarrow[\bowtie_j]{\varepsilon} & (\textbf{mkfun}\ (\textbf{send}\ \langle (\textbf{init}\ v_0\ p_j), \ldots, (\textbf{init}\ v_{p_j-1}\ p_j) \rangle)) \\
\textbf{send}\ \langle [v_0^0, \ldots, v_0^{p_j-1}], \ldots, [v_{p_j-1}^0, \ldots, v_{p_j-1}^{p_j-1}] \rangle & \xrightarrow[\bowtie_j]{\varepsilon} & \langle [v_0^0, \ldots, v_{p_j-1}^0], \ldots, [v_0^{p_j-1}, \ldots, v_{p_j-1}^{p_j-1}] \rangle
\end{array}
$$

$$
(\textbf{bsp\_p}\ ()) \xrightarrow[\delta_{i,j}]{\varepsilon} p_j
$$

where $\textbf{mkfun} = \textbf{apply}(\textbf{mkpar}(\textbf{fun}\ j\ t\ i \rightarrow \textbf{if}\ (\textbf{and}\ (\leq(0,i), <(i,p_j)))\ \textbf{then}\ (\textbf{access}\ t\ i)\ \textbf{else}\ \textbf{nc}))$

Figure 4: Parallel $\xrightarrow{\varepsilon}$-rules

The dynamic semantics is defined by an evaluation mechanism that relates expressions to values. To express this relation, we used a small-step semantics. It consists of a predicate between an expression and another expression defined by a set of axioms and rules called steps.

4.2 The small-steps dynamic semantics

The small-step semantics describes all the steps of the language from an expression to a value. We assume a finite set $\mathcal{N} = \{0, \ldots, P-1\}$ which represents the set of parallel machine names. We also assume finite sets $\mathcal{N}_j = \{0, \ldots, p_j - 1\}$ which represents the set of processor names of the parallel machine j.

The small-steps semantics has the following form: $e \rightharpoonup e'$. We note $\xrightarrow{*}$, for the transitive closure of $\rightharpoonup$, i.e., we note $e_0 \xrightarrow{*} v$ for $e_0 \rightharpoonup e_1 \rightharpoonup e_2 \rightharpoonup \ldots \rightharpoonup v$.

To define the relation $\rightharpoonup$, we begin to define three relations, one for each kind of expression: local (usual expression from a ML like language), parallel (from BSML) and departmental (from DMML). These relations are:

1. $e \underset{i,j}{\rightharpoonup} e'$ which could be read as "at processor i of the parallel machine j, the expression e is reduced to e'";
2. $e \underset{\bowtie_j}{\rightharpoonup} e'$ which could be read as "at parallel machine j, the expression e is reduced to e'";
3. $e \underset{\lightning}{\rightharpoonup} e'$ which could be read as "the expression e is reduced to e'" (by the whole metacomputer).

Each kind of expression, local, parallel or departmental contains usual function abstraction and application. Thus all these relations contain the following relation $\xrightarrow{\varepsilon}$, called the relation of "head reduction":

$$
\begin{array}{lcl}
(\textbf{fun}\ x \rightarrow e)\ v & \xrightarrow{\varepsilon} & e[x \leftarrow v] \\
(\textbf{let}\ x = v\ \textbf{in}\ e) & \xrightarrow{\varepsilon} & e[x \leftarrow v] \\
(\textbf{let rec}\ g\ x = v\ \textbf{in}\ e) & \xrightarrow{\varepsilon} & e[g \leftarrow (\textbf{fix}(\textbf{fun}\ x \rightarrow v))]
\end{array}
$$

We write $e_1[x \leftarrow e_1]$ the expression obtained by substituting all the free occurrences of x in e_1 by e_2. Free occurrences of a variable is defined as a classical and trivial inductive function on our

$$\begin{array}{rcl}
\textbf{mkdep}\ v & \underset{\delta}{\overset{\varepsilon}{\rightarrow}} & \lbag(v\ 0),\ldots,(v\ (P-1))\rbag \\
\textbf{applydep}\ \lbag v_0,\ldots,v_{P-1}\rbag\ \lbag v'_0,\ldots,v'_{P-1}\rbag & \underset{\delta}{\overset{\varepsilon}{\rightarrow}} & \lbag(v_0\ v'_0),\ldots,(v_{P-1}\ v'_{P-1})\rbag \\
\textbf{atdep}\ \lbag\ldots,v^m,\ldots\rbag\ m\ n & \underset{\delta}{\overset{\varepsilon}{\rightarrow}} & v^m_n \text{ where } v^m = \langle\ldots,v^m_n,\ldots\rangle
\end{array}$$

$$\begin{array}{rcl}
\textbf{get}\ \lbag g_0,\ldots,g_{P-1}\rbag\ \lbag g'_0,\ldots,g'_{P-1}\rbag & \underset{\delta}{\overset{\varepsilon}{\rightarrow}} & (\textbf{mkfun2}\ (\textbf{request}\ \lbag rq_0,\ldots,rq_{P-1}\rbag)) \\
\textbf{request}\ \lbag rq'_0,\ldots,rq'_{P-1}\rbag & \underset{\delta}{\overset{\varepsilon}{\rightarrow}} & (\textbf{received}\ \lbag rc_0,\ldots,rc_{P-1}\rbag) \\
\textbf{received}\ \lbag rc_0,\ldots,rc_{P-1}\rbag & \underset{\delta}{\overset{\varepsilon}{\rightarrow}} & (\textbf{return}\ \lbag rt_0,\ldots,rt_{P-1}\rbag) \\
\textbf{return}\ \lbag rt'_0,\ldots,rt'_{P-1}\rbag & \underset{\delta}{\overset{\varepsilon}{\rightarrow}} & \lbag r_0,\ldots,r_{P-1}\rbag
\end{array}$$

where $\forall j \in \{0\ldots P-1\}$ and $\forall i \in \{0\ldots p_j-1\}$

$$\begin{array}{rcl}
g_j & = & \langle v^j_0,\ldots,v^j_{p_j-1}\rangle \text{ and } g'_j = \langle v'^{ij}_0,\ldots,v'^{ij}_{p_j-1}\rangle \text{ and } rq_j = \langle rq^j_0,\ldots,rq^j_{p_j-1}\rangle \\
rq^j_i & = & (v'^{ij}_i, (\textbf{init}\ (\textbf{fun}\ clus \rightarrow (\textbf{init}\ (v^j_i\ clus)\ p_j))\ P)) \\
rq'_j & = & \langle rq'^{ij}_0,\ldots,rq'^{ij}_{p_j-1}\rangle \text{ and } rc_j = \langle rc^j_0,\ldots,rc^j_{p_j-1}\rangle \text{ and } rt_j = \langle rt^j_0,\ldots,rt^j_{p_j-1}\rangle \\
rq'^{ij}_i & = & (v'^{ij}_i, [[n^{(j,0)}_{(i,0)},\ldots,n^{(j,0)}_{(i,p_0-1)}],\ldots,[n^{(j,P-1)}_{(i,0)},\ldots,n^{(j,P-1)}_{(i,p_{P-1}-1)}]]) \\
rc^j_i & = & (v'^{ij}_i, [[n^{(0,j)}_{(0,i)},\ldots,n^{(0,j)}_{(p_0-1,i)}],\ldots,[n^{(P-1,j)}_{(0,i)},\ldots,n^{(P-1,j)}_{(p_{P-1}-1,i)}]]) \\
rt^j_i & = & (\textbf{init}\ (\textbf{fun}\ clus \rightarrow (\textbf{init}\ (\textbf{fun}\ pid \rightarrow \textbf{let}\ v = (\textbf{access}\ (\textbf{access}\ (\textbf{snd}\ rc^j_i)\ clus)\ pid) \\
 & & \qquad \textbf{in if}\ (\textbf{isnc}\ v)\ \textbf{then nc else}\ ((\textbf{fst}\ rc^j_i)\ v))) \\
rt'_j & = & \langle rt'^{ij}_0,\ldots,rt'^{ij}_{p_j-1}\rangle \text{ and } r_j = \langle r^j_0,\ldots,r^j_{p_j-1}\rangle \\
rt'^{ij}_i & = & [[v^{(0,j)}_{(0,i)},\ldots,v^{(0,j)}_{(p_0-1,i)}],\ldots,[v^{(P-1,j)}_{(0,i)},\ldots,v^{(P-1,j)}_{(p_{P-1}-1,i)}]] \\
r^j_i & = & [[v^{(j,0)}_{(i,0)},\ldots,v^{(j,0)}_{(i,p_0-1)}],\ldots,[v^{(i,P-1)}_{(i,0)},\ldots,v^{(j,P-1)}_{(i,p_{P-1}-1)}]]
\end{array}$$

and where

$$\begin{array}{l}
\textbf{mkfun2} = \textbf{applydep}(\textbf{mkpdep}(\textbf{fun}\ clus \rightarrow \textbf{fun}\ vec \rightarrow \textbf{apply}\ (\textbf{mkpar}\ (\textbf{fun}\ pid \rightarrow \textbf{fun}\ tab \rightarrow \\
\qquad \textbf{fun}\ cl \rightarrow \textbf{fun}\ pd \rightarrow \textbf{if}\ (\textbf{and}\ (\textbf{and}\ (\leq(0,cls),<(cl,P)),\textbf{and}\ (\leq(0,pd),<(pd,p_{cl}))))) \\
\qquad\qquad \textbf{then}\ (\textbf{access}\ (\textbf{access}\ tab\ cl)\ pd)\ \textbf{else nc}
\end{array}$$

Figure 5: Departmental $\overset{\varepsilon}{\rightarrow}$-rules

expressions. The **let rec** constructs use the operator **fix**:

$$\begin{array}{rcl}
\textbf{fix}(\textbf{fun}\ x \rightarrow\ e) & \underset{\delta}{\overset{\varepsilon}{\rightarrow}} & e[x \leftarrow \textbf{fix}(\textbf{fun}\ x \rightarrow\ e)] \\
\textbf{fix}(\textbf{op}) & \underset{\delta}{\overset{\varepsilon}{\rightarrow}} & \textbf{op}
\end{array}$$

The rules for operators are called δ-rules In the same manner, for usual constants and their operators we have some axioms given in Figure 6 (the figure shows only a subset of the rules).

Naturally, for the parallel and departmental constructors, we also have some reduction rules. Those rules are given in Figure 4 for the parallel expressions and in Figure 5 for the departmental expressions. Note that we have a special δ-rule (figure 4) for the **bsp_p** operator used in the scope of a parallel vector.

The evaluation of a **put** operator proceeds in two steps. First each processor creates a pure functional array of values. Thus we need a new kind of expressions, arrays: $[e,\ldots,e]$. **init** and

$(+\ (n_1, n_2))$	$\xrightarrow[\delta]{\varepsilon}$	n with $n = n_1 + n_2$	$(\mathbf{and}\ (b_1, b_2))$	$\xrightarrow[\delta]{\varepsilon}$	b
$(\mathbf{fst}\ (v_1, v_2))$	$\xrightarrow[\delta]{\varepsilon}$	v_1	$(\mathbf{snd}\ (v_1, v_2))$	$\xrightarrow[\delta]{\varepsilon}$	v_2
if true then e_1 **else**	$\xrightarrow[\delta]{\varepsilon}$	e_1	**if false then** e_1 **else** e_2	$\xrightarrow[\delta]{\varepsilon}$	e_2
$(\mathbf{isnc}\ \mathbf{nc})$	$\xrightarrow[\delta]{\varepsilon}$	**true**	$(\mathbf{isnc}\ v)$	$\xrightarrow[\delta]{\varepsilon}$	**false** if $v \neq \mathbf{nc}$
$(\mathbf{dm\_p}\ ())$	$\xrightarrow[\delta]{\varepsilon}$	**P**	$(\mathbf{dm\_bsp\_p}\ n)$	$\xrightarrow[\delta]{\varepsilon}$	$\mathbf{p}_n$

with $b = \mathbf{true}$ if $b_1 = \mathbf{true}$ and $b_2 = \mathbf{true}$ else $b = \mathbf{false}$

Figure 6: "Functional" δ-rules

access are used to manipulate these functional arrays of values:

$$\begin{array}{rcl} \mathbf{access}\ [v_0, \ldots, v_n, \ldots, v_l]\ n & \xrightarrow[\delta]{\varepsilon} & v_n \\ \mathbf{init}\ f\ l & \xrightarrow[\delta]{\varepsilon} & [(f\ 0), \ldots, (f\ (l\text{–}1))] \end{array}$$

In a second step, the **send** operation exchanges these arrays. For example the value at the index j of the array held at process i is sent to the process j and is stored at index i of the result. The function **mkfun** constructs a parallel vector of functions from the parallel vector of arrays.

For the **get** operator there are also several steps. Each processor of each BSP unit constructs its own **requests**. Then, it **received** the request of another processes, computes the sent values and then **return** them. At the end of the d-step, using an access to the matrix and the function **mkfun2** which constructs a departmental vector of parallel vectors of functions from the departmental vector of parallel vectors of arrays, we have on each component the construction of the result function from the **returned** values. Now, the complete definitions of our three kinds of reductions are:

$$\xrightarrow[i,j]{} = \xrightarrow{\varepsilon} \cup \xrightarrow[\delta]{\varepsilon} \quad \text{and} \quad \xrightarrow[\bowtie_j]{} = \xrightarrow[\bowtie_j]{\varepsilon} \cup \xrightarrow[\delta]{\varepsilon} \cup \xrightarrow{\varepsilon} \quad \text{and} \quad \xrightarrow[\lessapprox]{} = \xrightarrow[\lessapprox]{\varepsilon} \cup \xrightarrow[\delta]{\varepsilon} \cup \xrightarrow{\varepsilon}$$

It is easy to see that we cannot always make a head reduction.

4.3 Context rules

To define this deep reduction, we define some kind of contexts (an expression with a "hole" noted []) that have the abstract syntax given in Figure 7. They give the hole where the expression is reduced. The Γ context is used to define a "departmental reduction" of the meta-computer, i.e, a reduction outside a departmental vector, for example:

$$\Gamma = \mathbf{let}\ x = []\ \mathbf{in\ mkdep}\ (\mathbf{fun}\ clus \rightarrow \ldots)$$

The reduction will be occur at the hole to first compute the value of x. The Γ^j context is used to define in which parallel machine j of a departmental vector the reduction is done, i.e., which parallel machine j reduces its global expression. This context used the Γ_g context which defines a "global reduction" on a BSP unit. Note that, in this way, the hole is inside a departmental vector. For example, the following context: $\Gamma^j = \mathbf{applydep}\ v\ \langle\!|v_0, e_1, \ldots, \Gamma_g|\!\rangle$ and $\Gamma_g = \mathbf{mkpar}\ []$ is used to define that the last BSP unit first computes the argument of the **mkpar** operator.

After, the Γ_i^j context is used to define a local reduction at processor i on a parallel machine j: first the context finds a hole in a departmental vector, next, a Γ_i finds a hole in a parallel vector (which processor makes the reduction) and to end a Γ_l context finds the hole in a local expression,

i.e., standard OCaml expression. Note that this hole is inside a parallel vector which is inside a departmental vector.

Now we can reduce in-depth in the sub-expressions. To define this deep reduction, we use the inference rules of all the different kinds of context rules:

$$\frac{e \underset{i,j}{\rightharpoonup} e'}{\Gamma_i^j[e] \rightharpoonup \Gamma_i^j[e']} \qquad \frac{e \underset{\bowtie_j}{\rightharpoonup} e'}{\Gamma^j[e] \rightharpoonup \Gamma^j[e']} \qquad \frac{e \underset{\lightning}{\rightharpoonup} e'}{\Gamma[e] \rightharpoonup \Gamma[e']}$$

So we can reduce into the departmental and parallel vectors. In this way, the context gives the name of the processor and/or the name of the parallel machine where the expression is reduced. We can notice that the contexts give an order to evaluate an expression but not for the parallel and departmental vectors. These rules are not deterministic. Note that our kind of contexts used in the rules exclude each other by construction because the "hole" in a Γ_i^j context (resp. Γ^j context) is always in a component of a parallel vector (resp. departmental vector) and never for a Γ one. Thus, we have a rule and its context to reduce departmental expressions, global expressions and usual expressions (in the vectors) and we have the following result:

Theorem 1 (Confluence) *If* $e \overset{*}{\rightharpoonup} v_1$ *and* $e \overset{*}{\rightharpoonup} v_2$ *then* $v_1 = v_2$

Sketch of the Proof All the δ-rules ($\underset{\delta}{\rightharpoonup}$) and head reductions ($\rightharpoonup$, $\underset{\bowtie_j}{\rightharpoonup}$, $\underset{\lightning}{\rightharpoonup}$), i.e., the axioms, are deterministic (local, global and departmental ones). The rules are not always deterministic, i.e., several axioms can be applied at the same time, parallelism comes from the context rules. But if a context gives two possible reductions in a parallel or departmental vector, it is easy to see that these two reductions could be done in any order and give the same result because a reduction does not affect the result of the other one. So, the DMML language is confluent. Noted that our semantics shows a **received** operator which is immediately computed as a **return** operator: in the future we will design a distributed semantics proved correct with our semantics. This semantics needs to show the asynchronous *d*-steps of our language and, thus, the asynchronous reception and send off the messages (values).

5 Formal Costs

A formal cost model can be associated to reductions of the DMML language. "Cost terms" are defined, and each rule of the semantics is associated to a cost rule on cost terms. Given the *weak call-by-value*[1] *strategy* of section 4, a program is globally always reduced in the "same way". In this case, costs can be associated with terms rather that reductions. As stated in [22] "Each evaluation order has its advantages and disadvantages, but strict evaluation is clearly superior in at least one area: ease of reasoning about asymptotic complexity". It is the way we choose to ease the discussion about the compositional nature of the cost model of our language. No order of reduction is given between the different components of a parallel or departmental vector and their evaluations are done in parallel. The cost in this case is independent from the order of reduction. We noted $\mathcal{C}(e)$ the cost associated to an expression and $\mathcal{S}(v)$ the size of a serialized value v and $\bigoplus$ for the maximum of costs. We will not describe the cost of the evaluation of a local (i.e., functional) term: it is the same as a strict functional language (Objective Caml for example), but we give the costs of the evaluation of parallel and departmental operations. The costs associated to our programs follow our DMM model.

[1]Before evaluating an application, the function and its arguments need to be values and it is not possible to evaluate an expression which in under an abstraction

$$
\begin{array}{rcl}
\Gamma & ::= & [] \\
& | & \Gamma\ e \\
& | & v\ \Gamma \\
& | & \textbf{let}\ x = \Gamma\ \textbf{in}\ e \\
& | & \textbf{let rec}\ g\ x = \Gamma\ \textbf{in}\ e \\
& | & (\Gamma, e) \\
& | & (v, \Gamma) \\
& | & \textbf{if}\ \Gamma\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & \textbf{mkdep}\ \Gamma \\
& | & \textbf{applydep}\ \Gamma\ e \\
& | & \textbf{applydep}\ v\ \Gamma \\
& | & \textbf{get}\ \Gamma\ e \\
& | & \textbf{get}\ v\ \Gamma \\
& | & \textbf{atdep}\ \Gamma\ e_1\ e_2 \\
& | & \textbf{atdep}\ v_1\ \Gamma\ e_2 \\
& | & \textbf{atdep}\ v_1\ v_2\ \Gamma
\end{array}
\qquad
\begin{array}{rcl}
\Gamma^j & ::= & \Gamma^j\ e \\
& | & v\ \Gamma^j \\
& | & \textbf{let}\ x = \Gamma^j\ \textbf{in}\ e \\
& | & \textbf{let rec}\ g\ x = \Gamma^j\ \textbf{in}\ e \\
& | & (\Gamma^j, e) \\
& | & (v, \Gamma^j) \\
& | & \textbf{if}\ \Gamma^j\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & \textbf{mkdep}\ \Gamma^j \\
& | & \textbf{applydep}\ \Gamma^j\ e \\
& | & \textbf{applydep}\ v\ \Gamma^j \\
& | & \textbf{get}\ \Gamma^j\ e \\
& | & \textbf{get}\ v\ \Gamma^j \\
& | & \textbf{atdep}\ \Gamma^j\ e_1\ e_2 \\
& | & \textbf{atdep}\ v_1\ \Gamma^j\ e_2 \\
& | & \textbf{atdep}\ v_1\ v_2\ \Gamma^j \\
& | & \langle\!|e, \ldots, \overbrace{\Gamma_g}^{j}, e, \ldots, e|\!\rangle
\end{array}
\qquad
\begin{array}{rcl}
\Gamma_g & ::= & [] \\
& | & \Gamma_g\ e \\
& | & v\ \Gamma_g \\
& | & \textbf{let}\ x = \Gamma_g\ \textbf{in}\ e \\
& | & \textbf{let rec}\ g\ x = \Gamma_g\ \textbf{in}\ e \\
& | & (\Gamma_g, e) \\
& | & (v, \Gamma_g) \\
& | & \textbf{if}\ \Gamma_g\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & \textbf{mkpar}\ \Gamma_g \\
& | & \textbf{apply}\ \Gamma_g\ e \\
& | & \textbf{apply}\ v\ \Gamma_g \\
& | & \textbf{put}\ \Gamma_g \\
& | & \textbf{send}\ \Gamma_g \\
& | & \textbf{at}\ \Gamma_g\ e \\
& | & \textbf{at}\ v\ \Gamma_g
\end{array}
$$

$$
\begin{array}{rcl}
\Gamma_i^j & ::= & \Gamma_i^j\ e \\
& | & v\ \Gamma_i^j \\
& | & \textbf{let}\ x = \Gamma_i^j\ \textbf{in}\ e \\
& | & \textbf{let rec}\ g\ x = \Gamma_i^j\ \textbf{in}\ e \\
& | & (\Gamma_i^j, e) \\
& | & (v, \Gamma_i^j) \\
& | & \textbf{if}\ \Gamma_i^j\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & \textbf{mkdep}\ \Gamma_i^j \\
& | & \textbf{applydep}\ \Gamma^j\ e \\
& | & \textbf{applydep}\ v\ \Gamma_i^j \\
& | & \textbf{get}\ \Gamma_i^j\ e \\
& | & \textbf{get}\ v\ \Gamma_i^j \\
& | & \textbf{atdep}\ \Gamma_i^j\ e_1\ e_2 \\
& | & \textbf{atdep}\ v_1\ \Gamma_i^j\ e_2 \\
& | & \textbf{atdep}\ v_1\ v_2\ \Gamma_i^j \\
& | & \textbf{request}\ \Gamma_i^j \\
& | & \textbf{return}\ \Gamma_i^j \\
& | & \langle\!|e, \ldots, \overbrace{\Gamma_i}^{j}, e, \ldots, e|\!\rangle
\end{array}
\qquad
\begin{array}{rcl}
\Gamma_i & ::= & \Gamma_i\ e \\
& | & v\ \Gamma_i \\
& | & \textbf{let}\ x = \Gamma_i\ \textbf{in}\ e \\
& | & \textbf{let rec}\ g\ x = \Gamma_i\ \textbf{in}\ e \\
& | & (\Gamma_i, e) \\
& | & (v, \Gamma_i) \\
& | & \textbf{if}\ \Gamma_i\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & \textbf{mkpar}\ \Gamma_i \\
& | & \textbf{apply}\ \Gamma_i\ e \\
& | & \textbf{apply}\ v\ \Gamma_i \\
& | & \textbf{put}\ \Gamma_i \\
& | & \textbf{send}\ \Gamma_i \\
& | & \textbf{at}\ \Gamma_i\ e \\
& | & \textbf{at}\ v\ \Gamma_i \\
& | & \langle e, \ldots, \overbrace{\Gamma_l}^{i}, e, \ldots, e\rangle
\end{array}
\qquad
\begin{array}{rcl}
\Gamma_l & ::= & [] \\
& | & \Gamma_l\ e \\
& | & v\ \Gamma_l \\
& | & \textbf{let}\ x = \Gamma_l\ \textbf{in}\ e \\
& | & \textbf{let rec}\ g\ x = \Gamma_l\ \textbf{in}\ e \\
& | & (\Gamma_l, e) \\
& | & (v, \Gamma_l) \\
& | & \textbf{if}\ \Gamma_l\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & [\Gamma_l, e_1, \ldots, e_n] \\
& | & [v_0, \Gamma_l, \ldots, e_n] \\
& | & [v_0, v_1, \ldots, \Gamma_l, \ldots, e_n] \\
& | & [v_0, v_1, \ldots, v_{n-1}, \Gamma_l]
\end{array}
$$

Figure 7: Contexts of evaluation

The BSP cost of the parallel operations are given in Figure 8 for a BSP unit j. The parallel evaluation time of a parallel vector is the maximum of the sequential evaluation time of each of its components. Provided the two arguments of the parallel application are values (in this case, parallel vectors of values), the parallel evaluation time is the maximum of each point-wise application. The evaluation of a **put** operator requires a full super-step. To evaluate a **put** operator, first each

$$\mathcal{C}(\textbf{mkpar } e) \quad \leadsto \quad \mathcal{C}(e) + \bigoplus_{i=0}^{p_j-1} \mathcal{C}((f\ i)) \text{ if } e \xrightarrow{*} f$$

$$\mathcal{C}(\textbf{apply } e_1\ e_2) \quad \leadsto \quad \mathcal{C}(e_1) + \mathcal{C}(e_2) + \bigoplus_{i=0}^{p_j-1} \mathcal{C}((f_i\ v_i)) \text{ if } \begin{cases} e_1 \xrightarrow{*} \langle f_0, \ldots, f_{p_j-1} \rangle \\ e_2 \xrightarrow{*} \langle v_0, \ldots, v_{p_j-1} \rangle \end{cases}$$

$$\mathcal{C}(\textbf{put } e) \quad \leadsto \quad \mathcal{C}(e) + \bigoplus_{i=0}^{p_j-1} \sum_{j=0}^{p_j-1} \mathcal{C}((f_i\ j)) + \bigoplus_{i=0}^{p_j-1} (g_j \times h_i) + l_j \text{ where } \begin{cases} \text{if } e \xrightarrow{*} \langle f_0, \ldots, f_{p_j-1} \rangle \\ \text{if } (f_i\ m) \xrightarrow{*} v_m^i \\ \text{where } h_i = \sum_{m=0}^{p_j-1} \mathcal{S}(v_m^i) \end{cases}$$

$$\mathcal{C}(\textbf{at } e_1\ e_2) \quad \leadsto \quad \mathcal{C}(e_1) + \mathcal{C}(e_2) + (p_j - 1) \times \mathcal{S}(v_n) \times g_j + l_j \text{ if } \begin{cases} e_1 \xrightarrow{*} n \\ e_2 \xrightarrow{*} \langle v_0, \ldots, v_n, \ldots, v_{p_j-1} \rangle \end{cases}$$

Figure 8: Costs of our parallel operators

processor evaluates the p_j local values (first phase of the super-step). Once all values have been exchanged, a synchronization barrier occurs. At the beginning of this second super-step, each processor i constructs the function from the received values. So the total cost is the maximum of the sum of the computation time to compute the local values and to exchange them (send them and receive them). The evaluation of a global projection is: first the designed processor sends its value to all other processors and then a synchronization barrier occurs. The parallel evaluation time is thus the time to compute and send this value and the time of the synchronization barrier.

The DMM costs are given in Figure 9. We note ${}^c\langle\!\langle \mathcal{C}_0, \ldots, \mathcal{C}_P \rangle\!\rangle^c$ for the costs of each BSP unit (to follow our departmental cost model). Execution time for a complete program is thus bounded by: $\Psi = \bigoplus_{j=0}^{P-1} \mathcal{C}_j$ if $\mathcal{C}_j$ is the DMM cost of th BSP unit j. The parallel evaluation time of a departmental vector is the BSP costs of each BSP unit. Provided the two arguments of the departmental application are values (in this case, departmental vectors of values), the parallel evaluation time is the BSP costs of each component of each point-wise applications. The evaluation of a **atdep** operator requires a full d-step. The evaluation of a departmental projection is: the designed processor asynchronously sends its value to all other processors of all the BSP units. The departmental evaluation time is thus the time to compute e_0, e_1 and e_2 them for the BSP unit of the sent processor, the time to send this value to all other processors and for the other BSP unit, the time to asynchronously receive this value. The evaluation of a **get** operator also requires a full d-step. To evaluate a **get** operator, first each processor evaluates the $\sum_{j=0}^{P-1} p_j$ local integer (first phase of the d-step) to know to each BSP unit and processor it requires a value ("incoming partners"). Then, those values are asynchronously exchanged on the local network of the BSP units and on the WAN (thus the need of latency). To follow our departmental cost model, at the second phase of the d-step (communication phase), each BSP unit synchronizes with its "incoming partners" like in the definition of the inductive cost function Φ, the d-step cost of each BSP unit is the maximum of its own BSP and d-step computation with the computation of its "incoming partners".

The cost (parallel evaluation time) above are context independent. This is why our cost model is compositional. The compositional nature of this cost model relies on the absence of nesting of parallel and departmental vectors. Noted that our formal compositional cost model shows the evaluation of expressions. Of course, "abstract evaluations" could be done as a compositional static analysis [24] to have an approximation of the cost of the functional part of the program.

$$\mathcal{C}(\textbf{mkdep}\ e) \quad \leadsto \quad {}^c\!\langle\!|\mathcal{C}(e)+\mathcal{C}((f\ 0)),\ldots,\mathcal{C}(e)+\mathcal{C}((f\ (P-1)))|\!\rangle^c \text{ if } e \xrightarrow{*} f$$

$$\mathcal{C}(\textbf{applydep}\ e_0\ e_1) \quad \leadsto \quad {}^c\!\langle\!|\mathcal{C}_0^{e_0}+\mathcal{C}_0^{e_1}+\mathcal{C}(f_0\ v_0),\ldots,\mathcal{C}_{P-1}^{e_0}+\mathcal{C}_{P-1}^{e_1}+\mathcal{C}(f_{P-1}\ v_{P-1})|\!\rangle^c$$

$$\text{where} \begin{cases} \mathcal{C}(e_0) = {}^c\!\langle\!|\mathcal{C}_0^{e_0},\ldots,\mathcal{C}_{P-1}^{e_0}|\!\rangle^c \\ \mathcal{C}(e_1) = {}^c\!\langle\!|\mathcal{C}_0^{e_1},\ldots,\mathcal{C}_{P-1}^{e_1}|\!\rangle^c \end{cases} \text{and} \begin{cases} \text{if } e_0 \xrightarrow{*} \langle\!|f_0,\ldots,f_{P-1}|\!\rangle \\ \text{if } e_1 \xrightarrow{*} \langle\!|v_0,\ldots,v_{P-1}|\!\rangle \end{cases}$$

$$\mathcal{C}(\textbf{atdep}\ e_0\ e_1\ e_2) \quad \leadsto \quad {}^c\!\langle\!|\mathcal{C}_0,\ldots,\mathcal{C}_{P-1}|\!\rangle^c$$

$$\text{where} \begin{cases} \mathcal{C}(e_0) = {}^c\!\langle\!|\mathcal{C}_0^{e_0},\ldots,\mathcal{C}_{P-1}^{e_0}|\!\rangle^c \\ \mathcal{C}(e_1) = {}^c\!\langle\!|\mathcal{C}_0^{e_1},\ldots,\mathcal{C}_{P-1}^{e_1}|\!\rangle^c \\ \mathcal{C}(e_2) = {}^c\!\langle\!|\mathcal{C}_0^{e_2},\ldots,\mathcal{C}_{P-1}^{e_2}|\!\rangle^c \end{cases} \text{and} \begin{cases} \text{if } e_0 \xrightarrow{*} \langle\!|\langle v_0^0,\ldots,v_{p_0-1}^0\rangle,\ldots,\langle v_0^{P-1},\ldots,v_{p_{P-1}-1}^{P-1}\rangle|\!\rangle \\ \text{if } e_1 \xrightarrow{*} m \\ \text{if } e_2 \xrightarrow{*} n \end{cases}$$

and $\forall i \neq m$

$$\mathcal{C}_i = \bigoplus(C_i^{e_0}+C_i^{e_1}+C_i^{e_2}, C_m^{e_0}+C_m^{e_1}+C_m^{e_2}) + p_i \times \mathcal{S}(v_n^m) \times (g_m+G+g_i) + l_m + L + l_i$$

and $\mathcal{C}_m = C_m^{e_0}+C_m^{e_1}+C_m^{e_2}+((\sum_{j=0}^{P-1} p_j) \times \mathcal{S}(v_n^m) \times g_m) + l_m$

$$\mathcal{C}(\textbf{get}\ e_0\ e_1) \quad \leadsto \quad {}^c\!\langle\!|\mathcal{C}_0,\ldots,\mathcal{C}_{P-1}|\!\rangle^c$$

$$\text{where} \begin{cases} \mathcal{C}(e_0) = {}^c\!\langle\!|\mathcal{C}_0^{e_0},\ldots,\mathcal{C}_{P-1}^{e_0}|\!\rangle^c \\ \mathcal{C}(e_1) = {}^c\!\langle\!|\mathcal{C}_0^{e_1},\ldots,\mathcal{C}_{P-1}^{e_1}|\!\rangle^c \end{cases} \text{and} \begin{cases} \text{if } e_0 \xrightarrow{*} \langle\!|\langle f_0^0,\ldots,f_{p_0-1}^0\rangle,\ldots,\langle f_0^{P-1},\ldots,f_{p_{P-1}-1}^{P-1}\rangle|\!\rangle \\ \text{if } e_1 \xrightarrow{*} \langle\!|\langle k_0^0,\ldots,k_{p_0-1}^0\rangle,\ldots,\langle k_0^{P-1},\ldots,k_{p_{P-1}-1}^{P-1}\rangle|\!\rangle \end{cases}$$

and

$$\begin{aligned}
\mathcal{C}_i &= \bigoplus_{j\in\Omega_i}(C_i^{e_0}+C_i^{e_1}, C_j^{e_0}+C_j^{e_1}) + \text{Comm}_i \\
\Omega_i &= \{j \in \{0\ldots P-1\} \setminus \{i\} \ /\ \forall a \in \{0\ldots p_i-1\} \forall b \in \{0\ldots p_j-1\}\ (f_a^i\ j\ b) \xrightarrow{*} (\textbf{Some}\ n_b^j)\} \\
\text{Comm}_i &= \text{CommMP}_i + \text{CommBSP}_i + \sum_{i=0}^{P-1}\sum_{j=0}^{p_i-1} \mathcal{C}(f\ i\ j) + L \\
\text{CommMP}_i &= \sum_{j\in\Omega_i}((g_i+G+g_j)\times h_i^j + l_j + \sum_{a=0}^{p_j-1}\mathcal{C}((k\ n_a^j))) \text{ where } h_i^j = \sum_{a=0}^{p_j-1}\mathcal{S}(v_a^j) \text{ if } (k\ n_a^j) \xrightarrow{*} v_a^j \\
\text{CommBSP}_i &= g_i \times h + l_i + \sum_{a=0}^{p_i-1}\mathcal{C}((k\ n_a^i)) \text{ where } h = \sum_{j\in\Omega_i}\sum_{a=0}^{p_j-1}\mathcal{S}(v_a^j) \\
&\qquad \text{if } (f_a^i\ j\ a) \xrightarrow{*} (\textbf{Some}\ n_a^j) \text{ and } (k\ n_a^j) \xrightarrow{*} v_a^j
\end{aligned}$$

Figure 9: Cost of our departmental operators

With benchmarks [14] to have the execution time of the local reduction (for a functional language: application and primitive operations) and the whole parameters of the machines, an estimation of the execution time of the metacomputing program could easily be done (we would have a tool as described in [16]).

6 Implementation of the DMMLlib

There are two main versions of the DMMLlib library: a sequential version and a parallel one. Based on a confluent semantics, the evaluation of a pure functional parallel program will lead to the same value with both versions. In the sequential version, parallel and departmental vectors are implemented with Objective Caml arrays.

In the parallel version, our operations are implemented as SPMD programs. A parallel (resp. departmental) vector is supposed to contain one value per process (resp. per BSP unit). The

non-communicating operations are thus very simple to implemented using the "pid" of each BSP unit and each process.

To send values, we need first to "serialize" them. the module Marshal of Objective Caml provides functions to encode arbitrary data structures as sequences of bytes and to read and decode them back into the desired data structure.

Currenlty for the BSP part of our language, any MPI library can be used. In fact we only use a very small subpart of MPI: functions given the process identifier and the number of processes of the BSP unit and the all-to-all collective operations.

For the departmental part of our language, we use the thread facilities of the Objective Caml language: the communication environments of each process needs to save the functional value of its d-steps (each process as a variable which count the d-step of the process) and is thus implemented as a thread. The asynchronous requests and "return" of the **get** operator are also implemented as threads and use the TCP/IP facilities of the Objective Caml language (from the Unix module) to communicate the values.

But the implementation of DMMLlib is modular. The BSP part as well as the specific DMM operations are not directly implemented using MPI and TCP/IP functions and threads. They are implemented using respectively a module of type BSP_Comm and MPM_Comm which are partially given in figure 10. The missing functions are initialization, finalization, timing and aborting functions.

```
module type BSP_Comm =
 sig
  val pid : unit → int
  val nprocs : unit → int
  val send : α option array → α option array
 end

module type MPM_Comm =
 sig
  val cluspid : unit → int
  val nclus : unit → int
  val dstep : unit → int
  val store : α → unit
  val request : int → int → int → α
 end
```

Figure 10: BSP_Comm and MPM_Comm

The meaning of pid and nprocs are respectively the process identifier and the number of processes of the BSP unit. send takes on each process an array of size nprocs(). If at process j the value contained at index i is Some v then the value v will be send from process j to process i. If the value is None, nothing will be sent. In the result, which are also arrays, None at index j at process i means than the process j sent no value ot i and Some v means that process j sent the value v to i. A global synchronization occurs inside this communication function.

The meaning of cluspid and nclus are respectively the cluster identifier and the number of BSP units in the metacomputer. dstep give the current d-step. store stores the value given in argument in the communication environment of the process. This communication environment can be seen

as an association table between d-step and values. reset_dstep checks whether the d-step should be reset and the communication environments emptied.

7 More Examples and Experiments

7.1 Broadcast

In the broadcast program, a single process of a BSP unit, called the root r, sends a message to all other processes. It could be done in a direct way: each process of each BSP unit asks the value of the root (see example of section 3) and the cost is:

$$max \begin{cases} j \in \{0 \dots P-1\} \setminus \{r\} & (\mathcal{S}(v) \times p_j \times (g_j + G + g_r) + l_r + l_j + L \\ \text{and} \sum\limits_{i \in \{0 \dots P-1\}} & (g_r \times p_i \times \mathcal{S}(v)) + l_r) \end{cases}$$

where v is the sent value and $\mathcal{S}$ the size in bytes of the value. The cost of the program is the maximum time for a BSP unit to receive the value and for the root to send it to all the processes of all the BSP units.

Another way is that each process of each BSP unit receives from the root only a subpart of the message. Each BSP unit contains all the parts needed to rebuild the initial value. Then on each BSP unit there is a total exchange of these parts to obtain the whole message. Thus we have the following cost:

$$max \begin{cases} j \in \{0 \dots P-1\} \setminus \{r\} & (\mathcal{S}(v) \times (g_j + G + g_r) + l_r + l_j + L + (p_j \times g_j \times \lceil \frac{\mathcal{S}(v)}{p_j} \rceil + l_j) \\ \text{and } g_r \times P \times \mathcal{S}(v) + l_r + (p_j \times g_r \times \lceil \frac{\mathcal{S}(v)}{p_r} \rceil + l_r) \end{cases}$$

The cost of the program is the maximum time for a BSP unit to receive the parts of the value and to totally exchange them and for the root to send the parts and to totally exchange them. Note that this program uses a program to scatter the message from the root.

7.2 Departmental Reduction

Our second example is the classical parallel reduction: each process of each BSP unit contains a value and we want to obtain the sum of these values. For this, a naive algorithm could exchange all the needed values and then each process performs a local reduction. In this way, the cost of this program is close to the BSP cost of the direct algorithm.

As an example of a formal cost analysis (we refer to a report available at dmmllib.free.fr for more details about how to formally give costs to DMML programs) of a less naive algorithm, we choose the multiplication-reduction of polynomials: each process contains a polynomial and we want to compute their global multiplication. We make the following hypotheses: 1) The clusters are sorted by their efficiency to perform a BSP reduction 2) the coefficients of the polynomials ($\sum_{i=0}^{m} c_i X^i$) are stored in an array of floats such that c_i is located at position i. We write $\mathcal{S}(n)$ for the size of a polynomial of degree n. In this way, we have the following property: $\mathcal{S}(poly1 \times poly2) = \mathcal{S}(poly1) + \mathcal{S}(poly2)$ if we make the hypothesis that the size of a float does not depend of its value.

The algorithm runs as follow. First each BSP unit j performs a direct BSP reduction. The cost for each of them is thus:

$$A_j = (n \times p_j)^2 \times r_j + (p_j - 1) \times \mathcal{S}(n) \times g + l_j$$

where n is the maximal degree of the polynomials and r_j the time to perform a float multiplication. Second the root process of each BSP units receives the polynomials of the previous BSP unit. In this way, the cost to receive the polynomials is:

$$B_j = \sum_{\forall i<j} (l_i + (g_i + G + g_j) \times \mathcal{S}(p_j \times n) + l_j + L)$$

and to send them is:

$$C_j = l_j + \mathcal{S}(p_j \times n) \times g_j \times (P - j)$$

With these received polynomials, the BSP unit is able to finish its reduction and the cost is

$$D_j = (p_j - 1) \times (\sum_{\forall i<j} \mathcal{S}(p_i \times n) \times g_j + l_i + (\sum_{\forall i<j} \mathcal{S}(p_i \times n) \times r_j$$

The execution time for the program is thus:

$$\max_{j \in \{0 \ldots P-1\}} (A_j + B_j + C_j + D_j)$$

7.3 Benchmarks

Preliminary experiments have been done on a metacomputer with 6 Pentium IV nodes cluster interconnected with a Gigabit Ethernet network and with 3 Celeron III nodes cluster interconnected with a Fast Ethernet network. The two clusters are interconnected with a slow Ethernet network. Figure 11 summarizes the timings. These programs were run 10 times and the average was taken. The naive broadcast algorithm is clearly slower than the second algorithm. Preliminary experiments of parallel reduction of multiplication of polynomials have been done to show a performance comparison between a BSP algorithm on the metacomputer and DMM algorithms. The BSML program has been only run on the first cluster and contain the same number of polynomials: 3 processes contains 2 polynomials. Using a second cluster and a less naive algorithm achieved a scalability improvement.

8 Related Work

The asynchronous nature of some parallel patterns like farms and pipelines hampers their efficient implementation with flat data parallel languages with global barriers like in BSPlib library [14]. To overcome these limitations, the BSP PUB library [7] offers the use of oblivious synchronization and provides the capacity to partition the current BSP machine into several subsets. Each subset acts as an autonomous BSP computer with its own synchronizations. For example, divide-and-conquer parallel algorithms are a class of algorithms which seem to be difficult to write using the classical BSP model. Several models [9] allowing subset synchronization have been proposed to remedy to this problem. The authors of the BSP Worldwide Standard Library report claims that an unwanted consequence of group partitioning is a loss of accuracy of the associated performance model. Subset synchronizations are thus not suitable for metacomputing in general and in departmental metacomputing in particular. Another problem of this model is that the time of synchronization barriers does not change for the autonomous BSP computers and algorithms still have the same complexity than BSP algorithms. [18] presents a new parallel operator which allows us to divide the networks

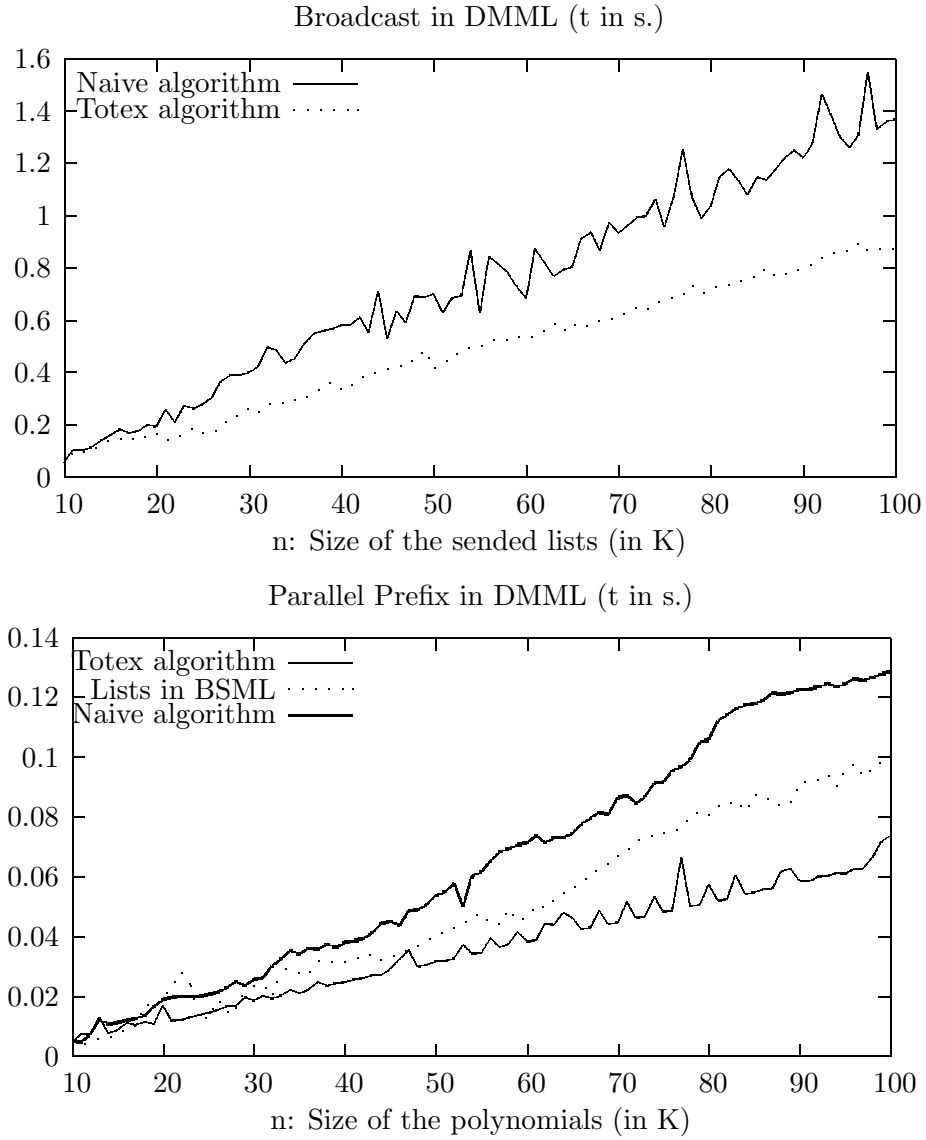

Figure 11: Benchmarks of collective operations

into subnetworks but still follows the "flat" BSP model which forbids subset synchronization. This new construction allows to write parallel divide-and-conquer algorithms.

Another feature of PUB is the oblivious synchronization. It is implemented through a function which does not return until all messages have been received for a given super-step. It implies that different processors can be in different super-steps at the same time and thus the MPM model of [26, 6] seems to be more adequate for a cost analysis of this kinds of programs. In our computation model, the two-tiered parallel levels are not based on subset synchronizations (it is a "flat two-tiered level model") and our cost model benefits of the advantages of the BSP and MPM models.

[30] presents an hierarchical extension of the BSP model with heterogeneous processors. But in this model, the execution of a program also proceeds in hyper-steps. Furthermore, the gateway is used for computation. The authors only analyze two-tiered level programs and we have the same problems as in the BSP^2 model: the time of the global barrier of synchronization of a hyper-step.

[8] also presents an hierarchical model (HiHCoHP for Hierarchical Hypercluster of Heterogeneous Processors) with heterogeneous processors with asynchronous steps (but without separate communication and computation phases) with network injection rate (as LogP), end-to-end bandwidth network latency, network capacity, message setup and message processing cost. The large number of parameters introduce a hardly tractable complexity. Moreover the programs are written in a low level language with neither a model of computation nor a semantics. The same problems occur in the model of [27].

[3, 4] also presents different models of computation but without a hierarchical extension of the BSP model or heterogeneous computers. Interesting work is the model of [5] for hierarchical and heterogeneous computers. But the main problem in this model (with also a low level language) is that the programs are difficult to analyze because the end-to-end bandwidth is combined with the network latency. In this framework a model of computation as in the BSP model lacks and deadlocks are possible.

Similar work to ours was conducted by [23], who performed an empirical study of the benefits of using a two-tiered parallel programming model. Their approach is based on data-duplication, all-to-all broadcasting and multicast message passing whereby these data would only be sent once between BSP units and then copied to all the BSP processors within the destination unit. In this way, they achieved a considerable scalability improvement. Another similar model is the pLogP model (parameterized Log-P) of [15]. The authors introduce a two-tiered extension of the Log-GP model [1] to optimize with the help of a cost analysis the collective operations of their own MPI library. But the authors do not present any formal semantics nor formal cost model and they use a low level language. To our knowledge, the DMML language is the first functional language for metacomputing with a formal semantics and a cost model.

9 Conclusions and Future Work

Earlier research has shown that many parallel applications can be optimized to run efficiently on hierarchical wide-area systems. The BSP model has proved to be a trusty and worthy tool in the discipline of parallel programming for producing reliable and portable codes with predictable efficiency. However, additional complexity introduced by metacomputing forces a review of the model. We have considered a hierarchical extension of the BSP model, called the DMM model and we have also described a new functional parallel language for this new model. This language is based on a formal semantics which is confluent and a formal cost model which allows cost analysis of the programs.

Several directions can be followed for future work. The first direction is the design of algorithms for the DMM model and their implementations with the DMMLlib library. To validate the cost model on these programs, we need a benchmark suite to determine the parameters of the metacomputer: this is an ongoing work. A complementary direction is to improve the current implementation of the DMMLlib library based on MPI and TCP/IP. In particular we will implement versions with adequate low level libraries for metacomputing such as [15, 2].

A semantical investigation of this framework is another direction of research. We will extend the type system of BSML [13] to DMML. We will also design a distributed semantics of DMML, closer to the implementation, and prove its correctness with respect to the semantics presented in the current paper. Future work will also consider fault tolerances as well as the study of interaction with non-purely functional features of Objective Caml.

Acknowledgments This work is supported by a grant from the French Ministry of Research and the ACI Grid program, under the project CARAML (www.caraml.org).

References

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, July 1997.

[2] O. Aumage, L. Bougé, and al. Madeleine II:A Portable and Efficient Communication Library for High-Performance Cluster Computing. *Parallel Computing*, 28(4):607–626, 2002.

[3] M. Banikazemi and D.K. Panda. Efficient collective communication on heterogeneous networks of workstations. Technical report, Ohio State University, 2000.

[4] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. *Math. Syst. Th.*, 27:431–452, 1994.

[5] P.B. Bhat, V.K. Prasanna, and C.S Raghavendra. Adaptative communication algorithms for distributed heterogeneous systems. *Parallel and Distributed Computation*, 59:252–279, 1999.

[6] V. Blanco, J. A. González, C. León, C. Rodríguez, G. Rodríguez, and M. Printista. Predicting the performance of parallel programs. *Parallel Computing*, 30:337–356, 2004.

[7] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library - Design, Implementation and Performance. In *Proc. of 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, San-Juan, Puerto-Rico, April 1999.

[8] F. Cappello, P. Fraigniaud, B. Mans, and A.L. Rosenberg. HiHCoHP toward a realistic communication model for hierarchical hyperclusters of heterogeneous processors. In *IEEE/ACM IPDPS'2001*. IEEE press, 2001.

[9] H. Cha and D. Lee. H-BSP: a Hierarchical BSP Computation Model. *Supercomputing*, 18(1):179–200, 2001.

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

[11] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):365–376, 2003.

[12] F. Gava. Design of Deparmental Metacomputing ML. In M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2004)*, LNCS, pages 50–53. Springer Verlag, 2004.

[13] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, number 2763 in LNCS, pages 215–229. Springer Verlag, 2003.

[14] J.M.D. Hill, W.F. McColl, and al. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.

[15] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, and R. F. H. Hofman. Network performance-aware collective communication for clustered wide-aera systems. *Parallel Computing*, 27:1431–1456, 2001.

[16] D. LeMétayer. ACE: an automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.

[17] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.07. Web pages at www.ocaml.org, 2003.

[18] F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. In H. Kosch, L. Boszormányi, and H. Hellwagner, editors, *Euro-Par 2003*, number 2790 in LNCS, pages 781–788. Springer Verlag, 2003.

[19] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.

[20] J. M. R. Martin and A. Tiskin. BSP modelling a two-tiered parallel architectures. In B. M. Cook, editor, *WoTUG'90*, pages 47–55, 1999.

[21] Z. Nemeth and V. Sunderam. Characterizing grids: Attributes, definitions, and formalisms. *Journal of Grid Computing*, 1:9–23, 2003.

[22] C. Okasaki. *Purely Functional Data-Structures*. Cambridge University Press, 1998.

[23] A. Plaat, H. E. Bal, and al. Sensitivity of Parallel Applications to large differences in bandwidth and latency in two-layer interconnects. *Futur Generation Computer Systems*, 2004. to appear.

[24] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *LISP and Functional Programming*, pages 65–78, 1994.

[25] D. Rémy. Using, Understanding, and Unravellling the OCaml Language. In G. Barthe, P. Dyjber, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, number 2395 in LNCS, pages 413–536. Springer, 2002.

[26] J. L. Roda, C. Rodríguez, D. G. Morales, and F. Almeida. Predicting the execution time of message passing models. *Concurrency: Practice and Experience*, 11(9):461–477, 1999.

[27] A.L. Rosenberg. Optimal sharing of partitionable workloads in heterogeneous networks of workstations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 413–419. CSREA Press, 2000.

[28] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[29] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[30] T. L. Williams and R. J. Parsons. Exploiting hierarchy in heterogeneous environments. In *IEEE/ACM IPDPS'2001*, pages 140–147. IEEE press, 2001.