Parallel Processing Letters

A FUNCTIONAL LANGUAGE FOR DEPARTMENTAL METACOMPUTING

FRÉDÉRIC GAVA and FRÉDÉRIC LOULERGUE
Laboratory of Algorithms, Complexity and Logic
Créteil, France; {gava,loulergue}@univ-paris12.fr

Received 16 September 2004
Revised 31 october 2004
Communicated by Sergei Gorlatch

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.

Keywords: Functional Programming, Bulk Synchronous Parallelism, 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* [18]. 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 ones. High-level language and *cost models* are needed to ease the programming of *hierarchical* architectures such as *clusters of clusters*.

Bulk-Synchronous Parallel ML or *BSML* is an extension of ML for programming *direct-mode* Bulk Synchronous Parallel (BSP) algorithms as functional programs. BSP computing is a parallel programming model introduced by Valiant [19] 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 [13].

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* [1]. 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 these limitations.

In [8] a previous version of the DMM cost model and the primitives of DMML were presented, but without a parallel implementation, a formal semantics, examples and experiments. A formal semantics is the most precise specification one could give for a new programming language. In our case it was a necessary very valuable guide for the parallel implementation. We also proved that the DMML language is deterministic.

Such a semantics is also a requirement for proving the correctness of programs. As usual in functional languages, we could *prove* the correctness of the DMML implementations of DMM algorithms with a *proof assistant* as done in [7] for BSP algorithms implemented in BSML. Using the *extraction* capability of proof assistants, we could generate a *certified* implementation of these algorithms. Nevertheless proofs of correctness of parallel programs (and also of the compilers and runtimes systems) go far beyond the scope of this paper and constitute the Propac project (wwwpropac.free.fr).

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 Deparmental Metacomputing ML or DMML (Section 3), a formal semantics (Section 4) for a core sub-language. Section 5 is devoted to two examples of collective communication operations and to the implementation. We discuss related work (Section 6) and conclude (Section 7).

2. A Model for Departmental Metacomputing

We assume throughout this paper that a *metacomputer* is a set of multiple clusters or parallel machines, i.e. a *cluster of clusters*, 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 WAN. 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 bandwidths.

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

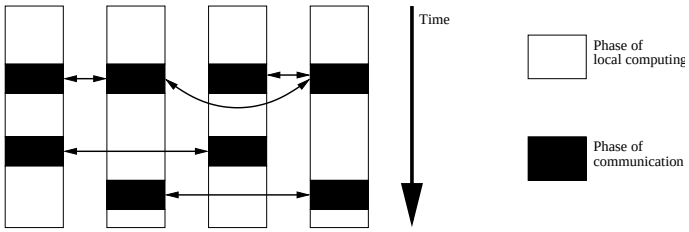

Figure 1: The MPM model of computation

conciseness, we refer to [17] for more details. In this model, a parallel computation is divided in *super-steps*, at the end of which exchanges of data and a synchronization barrier are 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: 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). g and l are expressed as multiples of the local processing speed s. 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 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 not 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 solution to the aforementioned problems. Starting from BSP, we address the problem of enlarging the number of parameters without introducing an unbearable complexity.

To remedy the first problem, [3] 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), black boxes are phases of communication. During these communication phases, the processes exchange the data they need for the next m-step.

To remedy the second problem, [14] investigated a two-level hierarchical BSP model for a cluster of SMP machines and two-levels of communications. 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 and then communicates 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 synchro-

nization 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*.

2.3. A Departmental Metacomputing Cost Model

However, 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. The result model of computation called DMM is introduced in this section.

A metacomputer in this model is thus characterized by the following parameters: P the number of parallel computers, L the latency of the departmental 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 unit; $\mathcal{S} = \{s_0, \ldots, s_{P-1}\}$ 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. A message sent from a processor within a parallel unit to another parallel unit first goes to the gateway using the local network, travels through 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 $\Omega_{d,i}$ of *incoming partners* for a BSP unit i and a d-step d which is the set of BSP units j which sent messages to BSP unit i during d-step d.

The execution time at the end of a d-step d at BSP unit i is written $\Phi_{d,i}$ and is inductively defined as:

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

for $i \in \{0, \ldots, P-1\}$ where $h^j_{d,i}$ denotes the number of words received by the BSP unit i from the BSP unit j during the d-step d 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:

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

bsp_p: unit → int **bsp_g**: unit → float **bsp_l**: unit → float
mkpar: (int → α) → α **par**
apply: (α → β) **par** → α **par** → β **par**
type α option = None | Some **of** α
put: (int → α option) **par** → (int → α option) **par** **at**: α **par** → int → α

Figure 2: The core BSMLlib library

where s_d^j is the number of super-steps of the BSP unit j during the d-step d and w_b^a the local processing time on processor b during each super-step a and $h_b^{(a)} = \max\{h_{b+}^{(a)}, h_{b-}^{(a)}\}$ where $h_{b+}^{(a)}$ (resp. $h_{b-}^{(a)}$) is the number of words transmitted (resp. received) by processor b during each super-step a. The execution time for a 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 its incoming partner and is therefore more accurate than the BSP^2 one: 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 (OCaml) [11]. In this section we describe the BSMLlib core and the new primitives added to obtain DMML.

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 [9]. The BSML parallel constructs operate 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 → 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*.

Asynchronous phases are programmed with **mkpar** and **apply**. The expression (**apply** (**mkpar** f) (**mkpar** e)) stores ((f i)(e i)) on process i. The communication and synchronization phases are expressed by **put**. Consider the expression: **put**(**mkpar**(**fun** i → fs_i)). 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, (fs_j i) must evaluate to None. This expression 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.

at is the synchronous projection primitive where (**at** vec n) returns 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:

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: The added primitives

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 [9]. 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 → x) and parfun f vv = apply (replicate f) vv
let bcast_direct rt vv = if (root<0 || root>=bsp_p()) then raise Bcast else
 let msg = mkpar (fun pid v dst → if pid=root then Some v else None)
  in parfun noSome (apply (put (apply msg vv)) (replicate root))
```

3.2. The DMML library

DMML extends BSMLlib by adding new primitives on a new level called departmental. The core of this library adds the primitives given in Figure 3. DMML 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, bandwidth 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.

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 OCaml value or a BSML value.

The DMML departmental constructs operate 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. The expression (**mkdep** f) is said to be *departmental*.

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

((f a)(e a)). Communications are expressed by **get**. Consider the expression:

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

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 $\mathsf{f}_{a,i}$ at process i of the BSP unit a must be such that ($\mathsf{f}_{a,i}\ b\ j$) evaluates to Some n. To receive no value ($\mathsf{f}_{a,i}\ b\ j$) must evaluate to None.

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

At process i of the BSP unit a ($\mathsf{f'}_{a,i}\ b\ j$) 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 primitive **atdep**. It is used in the same way as the **at** primitive 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 primitive should not be evaluated inside a **mkdep**. Using **atdep** the departmental 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 f v → apply f v)) gf) gv
let parfun_all f x = apply_all (replicate_all f) x

let get_one_all datas srcs =
 let send=parfun_all (fun v n → Some v) datas
 and n_srcs=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) cluster pid →
    if (cluster=a)&&(pid=i) then Some 0 else None) n_srcs in
 parfun2_all(fun f (a,i) → (noSome (f a i)))(get ask send) n_srcs

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

Note that in the above example, the four last lines only are specific to the direct broadcast algorithm. The remaining functions are part of the DMML standard library and they could and are used in many other cases.

Thus this example gives a taste of the DMML programming style. The communications functions are purely functional – which is a high level feature, not so common and a requirement to use the extraction capability of a proof assistant – but the explicit handling of messages could make them difficult to use and could seem not so high level. Nevertheless it is very easy, and concise, to build more and more complex functions, in particular using the higher-order nature of functions. The user of the DMML library is offered a complete set of functions (maps,

broadcasts, folds, scans, *etc.*) implemented with the primitives only and can still implement its own functions using the primitives. DMML programs can use the DMM parameters to adapt themselves to the underlying architecture, which make them portable (in the same way the BSP algorithms are portable).

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.

4.1. Syntax of the DMML core-language

Expressions, written e and variants, have the following abstract syntax:

$$\begin{array}{rcl} e & ::= & x \mid c \mid op \mid \textbf{fun}\ x \rightarrow e \mid (e\ e) \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e \\ & \mid & \textbf{mkpar}\ e \mid \textbf{apply}\ e\ e \mid \textbf{put}\ e \mid \textbf{at}\ e\ e \\ & \mid & \textbf{mkdep}\ e \mid \textbf{applydep}\ e\ e \mid \textbf{get}\ e\ e \mid \textbf{atdep}\ e\ e\ e \end{array}$$

In this grammar, x ranges over a countable set of identifiers. Constants c are () (the only value of type unit in OCaml), the integers, the booleans and the value **nc** (which stands for no communication) which plays the role of the None constructor in OCaml for the **put** communication primitive. The set of predefined operations *op* contains arithmetic and boolean operations, the test function **isnc** of the **nc** constant, the **fix** operator and the function of access to the parameters of the metacomputer.

This syntax is the programmer's syntax. Reduction can produce additional expressions, enumerated parallel vectors and enumerated departmental vectors:

$$e ::= \ldots \mid \langle\, e\,, \ldots,\ e\,, \ldots,\ e\,\rangle \mid \langle\!|e, \ldots, e, \ldots, e|\!\rangle$$

There is one semantics per size of the DMM machine. We will note a,b and variants, the BSP units, and i,j and variants the processors. By size we mean the number of BSP units, and p_a the number of processors of the BSP unit a, for each unit of the metacomputer. The size of departmental vectors is P and the sizes of parallel vectors can be any p_a.

Before presenting the dynamic semantics of our core-language, i.e, how the expresions are computed to values, we present the values themselves:

$$v \quad ::= \quad \textbf{fun}\ x\ \rightarrow e \mid c \mid op \mid \langle\, v\,, \ldots,\ v\,, \ldots,\ v\,\rangle \mid \langle\!|v, \ldots, v, \ldots, v|\!\rangle$$

4.2. Reduction rules

To express the DMML semantics, we use a *small-step* semantics. It consists of a predicate between an expression and another expression. The small-step semantics describes all the steps from an expression to a value. The small-steps semantics has the following form: $e \rightharpoonup e'$. We note $\stackrel{*}{\rightharpoonup}$, for the reflexive transitive closure of $\rightharpoonup$, e.g., we note $e_0 \stackrel{*}{\rightharpoonup} v$ 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 OCaml expressions), parallel (BSML expressions) and departmental (DMML expressions).

$$(\textbf{fun}\ x \to e)\ v \xrightarrow{\varepsilon} e[x \leftarrow v] \qquad (\textbf{let}\ x = v\ \textbf{in}\ e) \xrightarrow{\varepsilon} e[x \leftarrow v]$$

Figure 4: Head reductions

$$\begin{array}{rcl@{\qquad}rcl}
\textbf{fix}(\textbf{fun}\ x \to e) & \xrightarrow[\delta]{\varepsilon} & e[x \leftarrow \textbf{fix}(\textbf{fun}\ x \to e)] & \textbf{fix}(\textbf{op}) & \xrightarrow[\delta]{\varepsilon} & \textbf{op} \\
(\textbf{isnc}\ v) & \xrightarrow[\delta]{\varepsilon} & \textbf{false if } v \neq \textbf{nc} & (\textbf{isnc nc}) & \xrightarrow[\delta]{\varepsilon} & \textbf{true} \\
\textbf{if false then } e_1 \textbf{ else } e_2 & \xrightarrow[\delta]{\varepsilon} & e_2 & (\textbf{dm_bsp_p}\ a) & \xrightarrow[\delta]{\varepsilon} & \mathbf{p}_a \\
\textbf{if true then } e_1 \textbf{ else } e_2 & \xrightarrow[\delta]{\varepsilon} & e_1 & (\textbf{dm_p}\ ()) & \xrightarrow[\delta]{\varepsilon} & \mathbf{P} \\
\textbf{access}\ [v_0, \ldots, v_n, \ldots, v_l]\ n & \xrightarrow[\delta]{\varepsilon} & v_n & \textbf{init}\ l\ f & \xrightarrow[\delta]{\varepsilon} & [(f\ 0), \ldots, (f\ (l-1))]
\end{array}$$

Figure 5: Predefined operations

These three relations are :
$e \underset{i,a}{\rightharpoonup} e'$: at process i of the BSP unit a the expression e is reduced to e';
$e \underset{\bowtie_a}{\rightharpoonup} e'$: at parallel machine a, the expression e is reduced to e';
$e \underset{\Diamond}{\rightharpoonup} e'$: the expression e is reduced to e' by the whole metacomputer.

Each kind of expression, local, parallel or departmental contains usual function abstractions and applications. Thus all these relations contain the following relation $\xrightarrow{\varepsilon}$, called the relation of *head reduction* given in figure 4

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

Some rules, the δ-rules, for predefined operations are given in figure 5. Rules for parallel and departmental primitives and given respectively in figures 6 and 7.

We give here the semantics of the BSML primitive **put** in two steps, corresponding accurately to the current implementation of BSML. Firstly each processor creates a purely functional array of values by applying the function which it holds to all the possible numbers of processor in the unit where the reduction takes place. Then a lower level primitive **send** makes the exchanges and returns a parallel vector of arrays. The value at index j of the array is sent to the processor j if it is not **nc**. Processor j receives it and stores it at index i of result array. The function **mkf** builds the parallel vector of result functions from the parallel vector of arrays.

To do that we introduce new expressions : arrays, written $[e, \ldots, e, \ldots, e]$. The two predefined operations **init** and **access** operates on arrays (figure 5).

Rules of primitives **mkdep**, **applydep** and **atdep** are similar to the rules for BSML primitives but differ by the number of arguments and by kinds of vectors.

get needs, like **put**, two lower level primitives: **senddep** and **mkanswer**.

The first primitive (rule 10) adapts at the departmental level the **send** of the global level. The argument is a departmental vector of parallel vectors of arrays of arrays of values. The results is a value which has the "same type". The argument of this operation is such as at a given processor i of a unit a the array of arrays indicates for each pair (b, j) of unit and processor, the value to send to processor j of unit b. The result is such as each processor i of a unit a the array of arrays

$$\textbf{mkpar}\ v \quad \xrightarrow[\bowtie_a]{\varepsilon} \quad \langle (v\ 0), \ldots, (v\ (p_a - 1)) \rangle \tag{1}$$

$$\textbf{apply}\ \langle v_0, \ldots, v_{p_a-1} \rangle\ \langle v'_0, \ldots, v'_{p_a-1} \rangle \quad \xrightarrow[\bowtie_a]{\varepsilon} \quad \langle (v_0\ v'_0), \ldots, (v_{p_a-1}\ v'_{p_a-1}) \rangle \tag{2}$$

$$\textbf{at}\ \langle \ldots, v_n, \ldots \rangle\ n \quad \xrightarrow[\bowtie_a]{\varepsilon} \quad v_n \tag{3}$$

$$\textbf{put}\ \langle v_0, \ldots, v_i, \ldots, v_{p_a-1} \rangle \quad \xrightarrow[\bowtie_a]{\varepsilon} \quad (\textbf{mkf}\ (\textbf{send}\ \langle \ldots, (\textbf{init}\ p_a\ v_i), \ldots \rangle)) \tag{4}$$

$$\textbf{send}\ \langle [v_0^0, .., v_0^{p_a-1}], \ldots, [v_{p_a-1}^0, .., v_{p_a-1}^{p_a-1}] \rangle \quad \xrightarrow[\bowtie_a]{\varepsilon} \quad \langle [v_0^0, .., v_{p_a-1}^0], \ldots, [v_0^{p_a-1}, .., v_{p_a-1}^{p_a-1}] \rangle \tag{5}$$

where $\textbf{mkf} = \textbf{apply}\ (\textbf{mkpar}\ (\textbf{fun}\ a\ t\ i \rightarrow$
$\textbf{if}\ (0 \leq i) \& (i < (\textbf{dm\_bsp\_p}\ a))\ \textbf{then}\ (\textbf{access}\ t\ i)\ \textbf{else}\ \textbf{nc}))$

Figure 6: Reduction $\xrightarrow[\bowtie_a]{\varepsilon}$

indicates for each pair (b, j) of unit and processor, the value the processor j of unit b sent to processor i of unit a.

The second primitive (rule 11) takes as argument a departmental vector of parallel vectors of arrays of arrays of integers and value **nc** which indicates to the processor i of unit a, at the index j in array with the index b in the arrays of arrays, the number of the value requested by the processor j of the unit b to the processor i of the unit a. The values are given by the departmental vector of parallel vectors of functions, second argument of this primitive. The result is a departmental vector of parallel vectors of arrays of arrays of the values to be transmitted.

The rule 9 allows the following steps: 1) A departmental vector of parallel vectors of arrays of arrays indicating which numbers of values must be requested is created from the first argument of the **get**. It is the reduction of the expressions t_i^a which will create these arrays of arrays. 2) These numbers of values are transmitted to the concerned processors: it is the primitive **senddep** applied to the departmental vector describes at the preceding step which carries out this. 3) The values to be sent are prepared using the **mkanswer** primitive. 4) The values are sent with the second call to **senddep**. 5) Then the function **mkf2** transforms the departmental vector of parallel vectors of arrays of arrays into a departmental vector of parallel vectors of functions, result of the **get**.

The three relations are then obtained as follows:

$$\underset{i,a}{\rightharpoonup} = \xrightarrow{\varepsilon} \cup \xrightarrow[\delta]{\varepsilon} \quad \text{and} \quad \underset{\bowtie_a}{\rightharpoonup} = \xrightarrow[\bowtie_a]{\varepsilon} \cup \xrightarrow[\delta]{\varepsilon} \cup \xrightarrow{\varepsilon} \quad \text{and} \quad \underset{\Rightarrow}{\rightharpoonup} = \xrightarrow[\Rightarrow]{\varepsilon} \cup \xrightarrow[\delta]{\varepsilon} \cup \xrightarrow{\varepsilon}$$

4.3. Context rules

It is easy to see that we cannot always make a head reduction: we have to reduct within an expression. To define this *deep reduction*, we define some kind of *contexts*, i.e, an expression with a *hole* noted [] that have the abstract syntax given in Figure 8. The hole gives where expressions could be reduced. In this way, the contexts give the order of evaluation of the arguments of the constructions of the language, i.e, the strategie. We note **op** is a parallel or departmental primitive. The Γ context is used to define a *departmental reduction* of the metacomputer, i.e, a reduction

$$\textbf{mkdep}\ v \xrightarrow[\Leftrightarrow]{\varepsilon} \langle\!\|(v\ 0),\ldots,(v\ (P-1))\|\!\rangle \tag{6}$$

$$\textbf{applydep}\ \begin{array}{l}\langle\!\|v_0,\ldots,v_{P-1}\|\!\rangle\\ \langle\!\|v'_0,\ldots,v'_{P-1}\|\!\rangle\end{array} \xrightarrow[\Leftrightarrow]{\varepsilon} \langle\!\|(v_0\ v'_0),\ldots,(v_{P-1}\ v'_{P-1})\|\!\rangle \tag{7}$$

$$\textbf{atdep}\ \langle\!\|\ldots,\langle\ldots,v_i^a,\ldots\rangle,\ldots\|\!\rangle\ a\ i \xrightarrow[\Leftrightarrow]{\varepsilon} v_i^a \tag{8}$$

$$\textbf{get}\ \begin{array}{l}\langle\!\|\ldots,\langle\ldots,f_i^a\ \ldots\rangle,\ldots\|\!\rangle\\ \langle\!\|\ldots,\langle\ldots,g_i^a\ \ldots\rangle,\ldots\|\!\rangle\end{array} \xrightarrow[\Leftrightarrow]{\varepsilon} \begin{array}{l}(\textbf{mkf2}\,(\textbf{senddep}\,(\textbf{mkanswer}\\ \quad(\textbf{senddep}\langle\!\|\ldots,\langle\ldots,t_i^a\ldots\rangle,\ldots\|\!\rangle)\\ \quad\langle\!\|\ldots,\langle\ldots,g_i^a\ldots\rangle,\ldots\|\!\rangle)))\ \text{where}\\ t_i^a\ =\ (\textbf{init}\,P\,(\textbf{fun}\,b\rightarrow\\ \quad(\textbf{init}\,(\textbf{dm\_bsp\_p}\,b)\,(f_i^a\,b))))\end{array} \tag{9}$$

$$\textbf{senddep}\,\langle\!\|\ldots,\langle\ldots,t_i^a,\ldots\rangle,\ldots\|\!\rangle\ \text{where}\ t_i^a = \begin{bmatrix}[n_{(i,0)}^{(a,0)},\ldots,n_{(i,p_0-1)}^{(a,0)}],\\ \ldots,\\ [n_{(i,0)}^{(a,P-1)},\ldots,n_{(i,p_{P-1}-1)}^{(a,P-1)}]\end{bmatrix} \xrightarrow[\Leftrightarrow]{\varepsilon} \langle\!\|\ldots,\langle\ldots,t'^a_i,\ldots\rangle,\ldots\|\!\rangle\ \text{where}\ t'^a_i = \begin{bmatrix}[n_{(0,i)}^{(0,a)},\ldots,n_{(p_0-1,i)}^{(0,a)}],\\ \ldots,\\ [n_{(0,i)}^{(P-1,a)},\ldots,n_{(p_{P-1}-1,i)}^{(P-1,a)}]\end{bmatrix} \tag{10}$$

$$\textbf{mkanswer}\ \begin{array}{l}\langle\!\|\ldots,\langle\ldots,t_i^a\ldots\rangle,\ldots\|\!\rangle\\ \langle\!\|\ldots,\langle\ldots,g_i^a\ldots\rangle,\ldots\|\!\rangle\end{array} \xrightarrow[\Leftrightarrow]{\varepsilon} t'^a_i = \begin{array}{l}\langle\!\|\ldots,\langle\ldots,t'^a_i\ldots\rangle,\ldots\|\!\rangle\ \text{where}\\ (\textbf{init}\,P\,(\textbf{fun}\,b\rightarrow(\textbf{init}(\textbf{dm\_bsp\_p}\,b)\\ (\textbf{fun}\,j\rightarrow\textbf{let}\,v=\\ \quad(\textbf{access}(\textbf{access}\,t_i^a\,b)\,j)\ \textbf{in}\\ \quad\textbf{if}\,(\textbf{isnc}\,v)\ \textbf{then nc else}\,(g_i^a\,v)))))\end{array} \tag{11}$$

$$\text{where}\ \textbf{mkf2}\ =\ \begin{cases}\textbf{applydep}\,(\textbf{mkpdep}\,(\textbf{fun}\ a\rightarrow\\ \quad\textbf{apply}\,(\textbf{mkpar}\,(\textbf{fun}\ i\ t\ b\ j\rightarrow\\ \quad\quad\textbf{if}\,((0\leq b)\&(b<P)\&(0\leq j)\&(j<(\textbf{dm\_bsp\_p}\ b)))\\ \quad\quad\textbf{then}\ (\textbf{access}\,(\textbf{access}\ t\ b)\ j)\ \textbf{else nc}))))\end{cases}$$

Figure 7: Reduction $\xrightarrow[\Leftrightarrow]{\varepsilon}$

outside a departmental vector. For example:

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

The reduction will occur at the hole to first compute the value of x. The Γ^a context is used to define in which component of a departmental vector the reduction is done, i.e., which BSP unit a reduces its global expression. This context uses 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: $\Gamma^a = \textbf{applydep}\ v\ \langle\!\|v_0, e_1, \ldots, \Gamma_g|\!\rangle$ and $\Gamma_g = \textbf{mkpar}\ []$ is used to define that the last BSP unit first computes the argument of the **mkpar** primitive.

The Γ_i^a context is used to define a *local reduction* at processor i on a parallel machine a: first the context finds a hole in a departmental vector, next, a Γ_i context finds a hole in a parallel vector, i.e., 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:

$$
\begin{array}{lll}
\Gamma_i^a ::= \Gamma_i^a\ e & \Gamma^a ::= \Gamma^a\ e & \Gamma ::= [] \\
\quad \mid v\ \Gamma_i^a & \quad \mid v\ \Gamma^a & \quad \mid \Gamma\ e \\
\quad \mid \textbf{let}\ x = \Gamma_i^a\ \textbf{in}\ e & \quad \mid \textbf{let}\ x = \Gamma^a\ \textbf{in}\ e & \quad \mid v\ \Gamma \\
\quad \mid \textbf{if}\ \Gamma_i^a\ \textbf{then}\ e\ \textbf{else}\ e & \quad \mid \textbf{if}\ \Gamma^a\ \textbf{then}\ e\ \textbf{else}\ e & \quad \mid \textbf{let}\ x = \Gamma\ \textbf{in}\ e \\
\quad \mid \textbf{op}\ \Gamma_i^a\ e_2 \ldots e_n & \quad \mid \textbf{op}\ \Gamma^a\ e_2 \ldots e_n & \quad \mid \textbf{if}\ \Gamma\ \textbf{then}\ e\ \textbf{else}\ e \\
\quad \mid \textbf{op}\ v_1\ \Gamma_i^a \ldots e_n & \quad \mid \textbf{op}\ v_1\ \Gamma^a \ldots e_n & \quad \mid \textbf{op}\ \Gamma\ e_2 \ldots e_n \\
\quad \mid \ldots & \quad \mid \ldots & \quad \mid \textbf{op}\ v_1\ \Gamma \ldots e_n \\
\quad \mid \textbf{op}\ v_1\ v_2 \ldots \Gamma_i^a & \quad \mid \textbf{op}\ v_1\ v_2 \ldots \Gamma^a & \quad \mid \ldots \\
\quad \mid \langle\!|e, \ldots, \overbrace{\Gamma_i}^{a}, e, \ldots, e|\!\rangle & \quad \mid \langle\!|e, \ldots, \overbrace{\Gamma_g}^{a}, e, \ldots, e|\!\rangle & \quad \mid \textbf{op}\ v_1\ v_2 \ldots \Gamma
\end{array}
$$

$$
\begin{array}{lll}
\Gamma_i ::= \Gamma_i\ e & \Gamma_g ::= [] & \Gamma_l ::= [] \\
\quad \mid v\ \Gamma_i & \quad \mid \Gamma_g\ e & \quad \mid \Gamma_l\ e \\
\quad \mid \textbf{let}\ x = \Gamma_i\ \textbf{in}\ e & \quad \mid v\ \Gamma_g & \quad \mid v\ \Gamma_l \\
\quad \mid \textbf{if}\ \Gamma_i\ \textbf{then}\ e\ \textbf{else}\ e & \quad \mid \textbf{let}\ x = \Gamma_g\ \textbf{in}\ e & \quad \mid \textbf{let}\ x = \Gamma_l\ \textbf{in}\ e \\
\quad \mid \textbf{op}\ \Gamma_i\ e_2 \ldots e_n & \quad \mid \textbf{if}\ \Gamma_g\ \textbf{then}\ e\ \textbf{else}\ e & \quad \mid \textbf{if}\ \Gamma_l\ \textbf{then}\ e\ \textbf{else}\ e \\
\quad \mid \textbf{op}\ v_1\ \Gamma_i \ldots e_n & \quad \mid \textbf{op}\ \Gamma_g\ e_2 \ldots e_n & \quad \mid [\Gamma_l, e_2, \ldots, e_n] \\
\quad \mid \ldots & \quad \mid \textbf{op}\ v_1\ \Gamma_g \ldots e_n & \quad \mid [v_0, \Gamma_l, \ldots, e_n] \\
\quad \mid \textbf{op}\ v_1\ v_2 \ldots \Gamma_i & \quad \mid \ldots & \quad \mid \ldots \\
\quad \mid \langle e, \ldots, \overbrace{\Gamma_l}^{i}, e, \ldots, e\rangle & \quad \mid \textbf{op}\ v_1\ v_2 \ldots \Gamma_g & \quad \mid [v_0, v_1, \ldots, \Gamma_l]
\end{array}
$$

Figure 8: Contexts

$$
\frac{e \underset{i,a}{\rightharpoonup} e'}{\Gamma_i^a[e] \rightharpoonup \Gamma_i^a[e']} \qquad \frac{e \underset{\bowtie_a}{\rightharpoonup} e'}{\Gamma^a[e] \rightharpoonup \Gamma^a[e']} \qquad \frac{e \underset{\Rightarrow}{\rightharpoonup} e'}{\Gamma[e] \rightharpoonup \Gamma[e']}
$$

Proposition 1 *Let e be an expression. If $e \stackrel{*}{\rightharpoonup} v_1$ and $e \stackrel{*}{\rightharpoonup} v_2$ then $v_1 = v_2$.*

Sketch of the Proof. All the δ-rules and head reductions, 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. In this way the DMML language is confluent.□

5. Examples and Implementation

5.1. 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:

$$
max \left\{ \begin{array}{l} a \in \{0 \ldots P-1\} \setminus \{r\} \quad (\mathcal{S}(v) \times p_a \times (g_a + G + g_r) + l_r + l_a + L \\ \text{and} \quad \sum_{i \in \{0 \ldots P-1\}} (g_r \times p_i \times \mathcal{S}(v)) + l_r) \end{array} \right.
$$

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:

$$max \begin{cases} a \in \{0 \dots P\text{–}1\} \setminus \{r\} & (\mathcal{S}(v) \times (g_a + G + g_r) + l_r + l_a + L + (p_a \times g_a \times \lceil \frac{\mathcal{S}(v)}{p_a} \rceil + l_a) \\ \text{and } g_r \times P \times \mathcal{S}(v) + l_r + (p_a \times g_r \times \lceil \frac{\mathcal{S}(v)}{p_r} \rceil + l_r) \end{cases}$$

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.

5.2. 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 a performs a direct BSP reduction. The cost for each of them is thus:

$$A_a = (n \times p_a)^2 \times r_a + (p_a - 1) \times \mathcal{S}(n) \times g + l_a$$

where n is the maximal degree of the polynomials and r_a 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:

$$B_a = \textstyle\sum_{\forall b<a} (l_b + (g_b + G + g_a) \times \mathcal{S}(p_a \times n) + l_a + L)$$

and to send them is: $C_a = l_a + \mathcal{S}(p_a \times n) \times g_a \times (P - a)$.
With these received polynomials, the BSP unit is able to finish its reduction and the cost is $D_a = (p_a - 1) \times (\sum_{\forall b<a} \mathcal{S}(p_b \times n) \times g_a + l_b + (\sum_{\forall b<a} \mathcal{S}(p_b \times n) \times r_a$. The execution time for the program is thus: $\max_{a \in \{0 \dots P-1\}} (A_a + B_a + C_a + D_a)$.

5.3. Implementation of the DMML library

There are two main versions of the DMML 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 OCaml arrays.

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

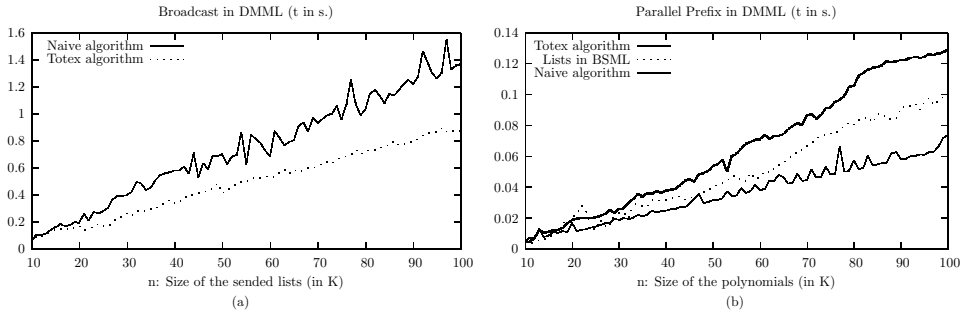

Figure 9: Benchmarks of collective operations

(resp. per BSP unit). The non-communicating primitives are thus very simple to implement using the "pid" of each BSP unit and each process.

Currently 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 functions.

For the departmental part of our language, we use the thread facilities of the OCaml language: communication environments [12] of each process are needed to save the functional value of its d-steps (each process as a variable which count the d-step) and is thus implemented as a thread. The asynchronous **request** and **return** of the **get** primitive are also implemented as threads and use the TCP/IP facilities of the OCaml language to communicate the values. The examples and a first implementation of DMML are available at dmmllib.free.fr.

5.4. 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 8 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.

6. Related Work

The asynchronous nature of some parallel patterns like farms and pipelines or divide-and-conquer parallel algorithms hampers their efficient implementation with flat data parallel languages with global barriers. To overcome these limitations, the BSP PUB library [4] provides the capacity to partition the current BSP machine into several subsets. Several models [6] allowing *subset synchronization* have been proposed. 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.

Another feature of PUB is the oblivious synchronization. It implies that different processors can be in different super-steps at the same time and thus the MPM model of [3] 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 BSP and MPM ones.

[20] 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 have the same problems as in the BSP^2 model: the time of the global barrier of synchronization of a hyper-step. [5] is another hierarchical model with heterogeneous processors and asynchronous steps. The large number of parameters in this model introduce a hardly tractable complexity. The same problems occur in the model of [16]. Interesting work is the model of [2] for hierarchical and heterogeneous computers. But the main problem in this model is that the programs are difficult to analyze because the end-to-end bandwidth is combined with the latency. Moreover, in all those frameworks an execution model as in the BSP one lacks and deadlocks are possible.

Similar work to ours was conducted by [15], 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, parametrized Log-P of [10]. The authors introduce a two-tiered extension of the Log-GP model 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.

7. 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 confluent semantics and allows programs cost analysis.

The first direction for future work is the design of algorithms for the DMM model and their implementations using the DMML library. To validate the cost model, we need a benchmark suite to determine the parameters of the metacomputer: this is an ongoing work. A complementary direction is to implement versions of DMML with adequate low level libraries for metacomputing [10,1]. Another direction of research

is the design of a distributed semantics of DMML, closer to the implementation, and the prove of its correctness with respect to the semantics presented here.

Acknowledgements This work is supported by a grant from the French Ministry of Research and the ACI Grid program (project CARAML). The authors wish to thank the anonymous referees for their comments. This paper is a revised version of the paper presented at CMPP'04.

References

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

[2] P.B. Bhat and al. Adaptative communication algorithms for distributed heterogeneous systems. *Parallel and Distributed Computation*, 59:252–279, 1999.

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

[4] O. Bonorder, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library. Web pages at http://www.uni-paderborn.de/~bsp/, 2004.

[5] F. Cappello, A.L. Rosenberg, and al. HiHCoHP toward a realistic communication model for hierarchical hyperclusters of heterogeneous processors. In *IPDPS'2001*. IEEE Press, 2001.

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

[7] F. Gava. Formal Proofs of Functional BSP Programs. *PPL*, 13(3):365–376, 2003.

[8] F. Gava. Design of Deparmental Metacomputing ML. In M. Bubak and al., editors, *ICCS 2004*, number 3038 in LNCS, pages 50–53. Springer Verlag, 2004.

[9] F. Gava and F. Loulergue. A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting. *FGCS*, 2004.

[10] 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.

[11] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System release 3.08.1. Web pages at www.ocaml.org, 2004.

[12] F. Loulergue, F. Gava, M. Arapinis, and F. Dabrowski. Semantics and Implementation of MSPML. *Int. Journal of Computer and Information Science*, 5(3), 2004.

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

[14] J. M. R. Martin and A. Tiskin. BSP modelling a two-tiered parallel architectures. In B. M. Cook, editor, *WoTUG'99*, pages 47–55, 1999.

[15] A. Plaat, H. E. Bal, and al. Sensitivity of Parallel Applications to large differences in bandwidth and latency in two-layer interconnects. *FGCS*, 2004.

[16] A.L. Rosenberg. Optimal sharing of partitionable workloads in heterogeneous networks of workstations. In *PDPTA'2000*, pages 413–419. CSREA Press, 2000.

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

[18] L. Smarr and C. E. Catlett. Metacomputing. *CACM*, 35(6):44–52, 1992.

[19] L.G. Valiant. A bridging model for parallel computation. *CACM*, 33(8):103, 1990.

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