Communication Primitives for Minimally Synchronous Parallel ML

Frédéric Loulergue

Laboratory of Algorithms, Complexity and Logic, Créteil, France
loulergue@univ-paris12.fr

Abstract. Minimally Synchronous Parallel ML is a functional parallel language whose execution time can then be estimated and dead-locks and indeterminism are avoided. Programs are written as usual ML programs but using a small set of additional primitives. It follows the cost model of the Message Passing Machine model (MPM). This paper explore two versions of an additional communication function: one uses this small set of primitives, the other one is considered as a primitive and implemented at a lower level.

1 Introduction

Bulk Synchronous Parallel ML (BSML) is a functional parallel language for the programming of Bulk Synchronous Parallel (BSP) [4] algorithms. It is an extension of the ML family of languages by a small set of parallel operations taken from a confluent extension of the λ-calculus. It is thus a deterministic and dead-lock free language.

We designed a new functional parallel language, without the synchronization barriers of BSML, called Minimally Synchronous Parallel ML (MSPML) [1]. As a first phase we aimed at having (almost) the same source language and high level semantics (programming view) than BSML (in particular to be able to use with MSPML work done on type system and proof of parallel BSML programs), but with a different (and more efficient for unbalanced programs) low-level semantics and implementation.

MSPML will also be our framework to investigate extensions which are not suitable for BSML, such as the nesting of parallel values or which are not intuitive enough in BSML, such as spatial parallel composition. We could also mix MSPML and BSML for meta-computing. Several BSML programs could run on several parallel machines and being coordinated by a MSPML-like program.

MSPML Programs are written as usual ML programs but using a small set of additional functions. Provided functions are used to access the parameters of the parallel machine and to create and operate on a parallel data structure. This paper explore the writing of an additional communication function using this small set of primitives. This function could also be considered as a primitive. These two versions are compared.

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3036, pp. 401–404, 2004.

2 Minimally Synchronous Parallel ML

BSPWB, for *BSP Without Barrier*, is a model directly inspired by the BSP model [4]. It proposes 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. During this communication phase the processes exchange the data they need for the next m-step.

The parallel machine in this model is characterized by three parameters (expressed as multiples of the processors speed): the number of processes p, the latency L of the network, the time g which is taken to one word to be exchanged between two processes. This model could be applied to MSPML but it will be not accurate enough because the bounds used in the cost model are too coarse.

The Message Passing Machine model (MPM) [3] gives a better bound $\Phi_{s,i}$. The parameters of the Message Passing Machine are the same than those of the BSPWB model. The model uses the set $\Omega_{s,i}$ for a process i and a m-step s defined as:

$$\Omega_{s,i} = \{j/\text{process } j \text{ sends a message to process } i \text{ at m-step } s\} \cup \{i\}$$

Processes included in $\Omega_{s,i}$ are called "incoming partners" of process i at m-step s. $\Phi_{s,i}$ is inductively defined as:

$$\begin{cases} \Phi_{1,i} = & \max\{w_{1,j}/j \in \Omega_{1,i}\} + (g \times h_{1,i} + L) \\ \Phi_{s,i} = & \max\{\Phi_{s-1,j} + w_{s-1,j}/j \in \Omega_{s,i}\} + (g \times h_{s,i} + L) \end{cases}$$

where $h_{s,i} = \max\{h^+_{s,i}, h^-_{s,i}\}$ for $i \in \{0, \ldots, p-1\}$ and $s \in \{2, \ldots, R\}$. Execution time for a program is thus bounded by: $\Psi = \max\{\Phi_{R,j}/j \in \{0, 1, \ldots, p-1\}\}$.

The MPM model takes into account that a process only synchronizes with each of its incoming partners and is therefore more accurate. The MPM model is used as the execution and cost model for our Minimally Synchronous Parallel ML language.

There is no implementation of a full Minimally Synchronous Parallel ML (MSPML) language but rather a partial implementation as a library for the functional programming language Objective Caml [2] (using TCP/IP for communications). The so-called MSPML library is based on the following elements:

```
bsp_p: unit->int            apply: ('a->'b) par->'a par->'b par
mkpar: (int->'a)->'a par    get: 'a par->int par->'a par
```

It gives access to the parameters of the underling architecture which is considered as a Message Passing Machine (MPM). In particular, **p()** is p, the static number of processes of the parallel machine. The value of this variable does not change during execution. There is also an abstract polymorphic type **'a par** which represents the type of p-wide parallel vectors of objects of type **'a**, one per process. The nesting of **par** types is prohibited. This can be ensured by a type system .

The parallel constructs of MSPML 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 **fun pid-¿e** for **f** 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*.

In the MPM model, an algorithm is expressed as a combination of asynchronous local computations and phases of communication. Asynchronous phases are programmed with **mkpar** and the point-wise parallel application **apply** which is such as **apply(mkpar f)(mkpar e)** stores **(f i)(e i)** on process i.

The communication phases are expressed by **get**. Its semantics is given by:

$$\texttt{get}\ \langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle\ \langle\ i_0\ ,\ldots,\ i_{p-1}\ \rangle = \langle\ v_{i_0 \% p}\ ,\ldots,\ v_{i_{(p-1)\%p}}\ \rangle$$

When **get** is called, each process i stores the value v_i in its communication environment. This value can then be requested by a process j which arrived at a later m-step.

3 A New Communication Function

The communication function described below uses functions from the MSPML standard library (http://mspml.free.fr) and some sequential functions. Their semantics follow:

$$\begin{array}{ll} \textbf{ft}\ n_1\ n_2 = [n_1; n_1+1; \ldots; n_2] & \textbf{procs}() = [0, \ldots, p()] \\ \quad \textbf{fill}\ e\ [x_1; \ldots; x_n]\ m = [x_1; \ldots; x_n; \underbrace{e; \ldots; e}_{m-n}] & \\ \quad \textbf{nfirst}\ n\ [x_1; \ldots\ x_n] = [x_1; \ldots; x_k] \text{ where } k = \min\{m, n\} & \end{array}$$

The **get_list** function is similar to **get** but it takes as second argument a parallel vector of lists of integers rather than a parallel vector of integers. Its semantics is thus given by:

$$\textbf{get\_list}\langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle\langle [i_1^0; \ldots; i_{k_0}^0], \ldots, [i_1^j; \ldots; i_{k_j}^j], \ldots, [i_1^{p-1}; \ldots; i_{k_{p-1}}^j]\rangle$$
$$= \langle\ [v_{i_1^0}; \ldots; v_{i_{k_0}^0}]\ ,\ldots,\ [v_{i_1^j}; \ldots; v_{i_{k_j}^j}]\ ,\ldots,\ [v_{i_1^{p-1}}; \ldots; v_{i_{k_{p-1}}^j}]\ \rangle$$

The problem is that the lists may not have the same length. Thus we will proceed in two phases. First we define an auxiliary **get_list_sl** function which takes a third argument: the length of the lists which are assumed to have the same length. Then we use it to define the general **get_list** function:

```
let rec get_list_sl vv vl = function    0 -> replicate []
  | n -> let vh=parfun List.hd vl and vt=parfun List.tl vl in
    let vg = get vv vh in
      parfun2 (fun h t->h::t) vg (get_list_sl vv vt (n-1))
let get_list vv vl =
  let vlen=parfun List.length vl in let mlen=reduce max vlen in
  let vl2=apply2 (mkpar fill) vl mlen in
  parfun2 nfirst vlen (get_list_sl vv vl2 (unsafe_proj mlen))
```

The **reduce** function has the following semantics:

$$\textbf{reduce}\ \oplus\ \langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle\ =\ \langle\ \oplus_{0\le k<p} v_k\ ,\ldots,\ \oplus_{0\le k<p} v_k\ ,\ldots,\ \oplus_{0\le k<p} v_k\ \rangle.$$

The **get_list** function implemented using the **get** primitive has MPM cost given by the following formula: $n + t_{\text{reduce}} + n \times (s \times g + L)$ where n is the length of the biggest list, s is the size of each element of the lists and t_{reduce} is the time required to compute the maximum length. For a direct **reduce** function $t_{\text{reduce}} = p + 2 \times (g \times (p-1) + L)$.

The low level implementation does not need the computation of the maximum length. Furthermore it is possible to use threads for the requests: a process will send sequentially its requests to the processes given by the lists without waiting for the answers. Thus the cost formula is: $L + n_i \times s \times g$ plus an additional overhead introduced by the use of the threads. This formula is given for a process i where n_i is the length of the integer list at process i.

We performed tests to compare the two **get_list** using the following programs:

```
let mshift d l v=get_list v (mkpar(fun i->ft (i+d) (i+d+l)))
let pre n v=get_list v (mkpar(fun i->ft n (n+1+(nmod(i-n)(p()))))))
```

We performed the tests on a cluster of 11 Pentium III processors with a fast Ethernet dedicated network. Let summarizes the results where the values are the average (10 tests from 2 to 11 processors were run) efficiency of the high-level implementation with respect to the low-level implementation. For the **mshift** function the ratio range from 20% to 60% for size between 1 and 1000. For sizes greater than 10K the efficiency is almost the same for the two versions. For the **pre** function the ratio is between 15% and 70%. The advantage of the primitive decreases with the size but the asynchronous nature of the function makes the advantage still interesting.

4 Conclusions and Future Work

We have explored how to write communication functions for the Minimally Synchronous Parallel ML language using only the unary **get** communication primitive: the **get_list** function allows to receive messages from several processes. It could also be considered as a communication primitive: the low-level parallel implementation described in this paper follows the execution model of the Message Passing Model. This implementation is more efficient but the proof of correctness is not done, while it is simple for the first version.

References

1. M. Arapinis, F. Loulergue, F. Gava, and F. Dabrowski. Semantics of Minimally Synchronous Parallel ML. In W. Dosch and R. Y. Lee, editors, *SNPD'03*, pages 260–267. ACIS, 2003.
2. Xavier Leroy. The Objective Caml System 3.07, 2003. web pages at www.ocaml.org.
3. 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.
4. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.