Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction

Frédéric Loulergue, Frédéric Gava, and David Billiet

Laboratory of Algorithms, Complexity and Logic – University Paris XII
61, avenue du Général de Gaulle – 94010 Créteil cedex – France
http://bsmllib.free.fr

Abstract. BSML is a library for parallel programming with the functional language Objective Caml. It is based on an extension of the λ-calculus by parallel operations on a parallel data structure named parallel vector. The execution time can be estimated, dead-locks and indeterminism are avoided. Programs are written as usual functional programs (in Objective Caml) 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 parallel vectors. It follows the execution and cost model of the Bulk Synchronous Parallel model. The paper presents the lastest implementation of this library and experiments of performance prediction.

1 Introduction

The design of parallel programs and parallel programming languages is a trade-off. On one hand the programs should be efficient. But the efficiency should not come at the price of non portability and unpredictability of performances. The portability of code is needed to allow code reuse on a wide variety of architectures and to allow the existence of legacy code. The predictability of performances is needed to guarantee that the efficiency will always be achieved on any architecture.

Another very important characteristic of parallel programs is the complexity of their semantics. Deadlocks and indeterminism often hinder the practical use of parallelism by a large number of users. To avoid these undesirable properties, a trade-off has to be made between the expressiveness of the language and its structure which could decrease the expressiveness.

Bulk Synchronous Parallelism [15, 1] (BSP) is a model of computation which offers a high degree of abstraction like PRAM models but yet a realistic cost model based on a structured parallelism: deadlocks are avoided and indeterminism is limited to very specific cases in the BSPlib library [8]. BSP programs are portable across many parallel architectures.

Our research aims at combining the BSP model with functional programming. We obtained the Bulk Synchronous Parallel ML language (BSML) based on a *confluent* extension of the λ-calculus. Thus BSML is deadlock free and deterministic. Being a high-level language, programs are easier to write, to reuse and to

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3515, pp. 1046–1054, 2005.
© Springer-Verlag Berlin Heidelberg 2005

compose. It is even possible to *certify* the correctness of BSML programs [5] with the help of the Coq proof assistant. The performance prediction of BSML programs is possible. BSML has been extended in many ways throughout the years and the papers related to this research are available at `http://bsml.free.fr`.

In section 2 we present the core of BSML, which is in fact currently a library, the BSML library, for the Objective Caml language [12]. Section 3 gives an overview of the new modular implementation of the current BSML library. Performance prediction is considered in section 4: we first describ how the BSP parameters of a parallel machine could be benchmarked and then we present a small experiment. Related work is presented in section 5. We conclude in section 6.

2 The BSML Library

There is currently no implementation of a full BSML language but rather a partial implementation as a library for Objective Caml language [12]. We assume the reader has basic knowledge about functional programming with Objective Caml and about the Bulk Synchronous Parallel model [1].

2.1 Primitives

BSML does not rely on SPMD programming. Programs are usual "sequential" Objective Caml programs but work on a parallel data structure. Some of the advantages is a simpler semantics and a better readability: the execution order follows (or at least the results is such as the execution order seems to follow) the reading order.

The core of the BSML library is based on the following elements:

bsp_p: unit→int
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

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 [6].

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→e to show that the expression e may be different on each processor. This expression e is said to be *local*. The expression (**mkpar** f) is a parallel object and it is said to be *global*.

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

The expression (**apply** (**mkpar** f) (**mkpar** e)) stores ((f i)(e i)) on process i.

Let consider the following expression:

```
let vf = mkpar(fun i→(+) i) and vv = mkpar(fun i→2*i+1) in
apply vf vv
```

The two parallel vectors are respectively equivalent to:

fun x→x+0	**fun** x→x+1	$\cdots$	**fun** x→x+(p−1)

and

0	3	$\cdots$	$2 \times (p-1)+1$

The expression **apply** vf vv is then evaluated to:

0	4	$\cdots$	$2 \times (p-1)+2$

Readers familiar with BSPlib [8] 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:

put(**mkpar**(**fun** i→fs_i)) (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, (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 synchronous projection operation **at** is such as (**at** vec n) return the nth value of the parallel vector vec. **at** expresses communication and synchronization phases. The projection should not be evaluated inside the scope of a **mkpar**. This is enforced by our type system [6].

For the pure functional subset of Objective Caml, BSML is also purely functional: there are *no side-effects*. Moreover these four parallel functions are called *primitives* because they need lower-level libraries (C libraries wrapped to Objective Caml code) to be implemented. In the modular version of BSML the implementation of these functions rely either on MPI [14], PUB [2], PVM [7] or on the TCP/IP functions provided by the Unix module of Objective Caml.

2.2 Examples

Other functions are very often used when one write BSML programs. Nevertheless these functions can be defined using *only* the primitives and require no lower-level libraries. They are part of what is called the standard library of BSML.

For example the function replicate creates a parallel vector which contains the same value everywhere.

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

It is also very common to apply the same sequential function at each process. It can be done using the parfun functions: they differ only in the number of arguments of the function to apply:

```
let parfun f v = apply (replicate f) v
let parfun2 f v1 v2 = apply (parfun f v1) v2
```

The semantics of the total exchange function is given by:

$$\text{totex}\ \langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle\ =\ \langle\ f\ ,\ldots,\ f\ ,\ldots,\ f\ \rangle$$

where f is such as $\forall i.(0 \leq i < p) \Rightarrow (f\ i) = v_i$. The code is as follows where compose is usual composition of functions:

```
(* totex: α par →(int →α ) par *)
let totex vv = parfun(compose noSome)(put(parfun (fun v dst→Some v) vv))
```

Its parallel cost is $(p-1) \times s \times g\ +\ L$, where s denotes the size of the biggest value v held at some process n in words.

From it we can obtain a version which returns a parallel vector of lists:

```
(* totex_list: α par→α  list par *)
let totex_list v = parfun2 List.map (totex v) (replicate(procs()))
```

where List.map $f\ [v_0;\ldots;v_n]\ =\ [(f\ v_0);\ldots;(f\ v_n)]$ and procs() $=\ [0;\ldots;p-1]$.

In section 4 we will use the following parallel reduction function:

```
let fold_direct f vec = let fold h::t=List.fold_left f h t in
parfun fold (totex_list vec)
```

where List.fold_left is a sequential reduction on lists in the Objective Caml standard library. Its BSP cost formula is (assuming op has a small and constant cost): $2 \times p - 1 + (p-1) \times s \times g + L$.

3 An Overview of the Implementation

In the implementation of the BSML library version 0.3, the Bsmllib module which contains the primitives presented in section 2.1, is implemented in SPMD style using a lower level communication library. This module called Comm is based on the following main elements:

```
pid: unit →int    nprocs: unit →int    send: α option array →α option array
```

There are several implementations of Comm based on MPI [14], PVM [7], BSPlib [8], PUB [2], TCP/IP, etc. The implementation of all the other modules of the BSML library, including the core module, is independent on the actual implementation of Comm, but only depends on its interface.

The meaning of pid and nprocs is obvious, they give respectively the process identifier and the number of processes of the parallel machine. send takes on each process an array of size nprocs(). These arrays contain optional values. The input values are obtained by applying at each process i, the function f_i (argument of the **put** primitive) to integers from 0 to (nprocs()−1).

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 the None value, 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 a non (Some v) value means that process j sent the value v to i. A global synchronization occurs inside this communication function. **put** and **at** are implemented using send.

The implementation of the abstract type, **mkpar** and **apply** is as follows:

type α **par** $= \alpha$ **let mkpar** f = f (pid()) **let apply** f v = f v

For the MPI version, the pid function does not call the MPI_Comm_rank function each time. The MPI function is called only one time at initialization and the value is store in a reference. The pid function only obtains the value of this reference.

Let see how the **put** primitive works. Consider a parallel machine with 4 processors and functions f_i whose types are int→α **par** such as $(f_i\ (i+1)) =$ Some v_i for $i = 0, 1, 2$ and $(f_i\ j) =$ None otherwise.

The expression **mkpar(fun** i →f_i) would be evaluated as follows:

1. First at each process the function is applied to all process identifiers to produce p values, the messages to be sent by the processes. In the following figure, on the left side, a column represents the values produced at one process and the lines are ordered by destination (first line represents the messages to be sent to process 0, etc.).

2. Then the exchange of messages is actually performed. If we think of the table as a matrix, the resulting matrix is obtained by transposition (right side):

None	None	None	None
Some v_0	None	None	None
None	Some v_1	None	None
None	None	Some v_2	None

None	Some v_0	None	None
None	None	Some v_1	None
None	None	None	Some v_2
None	None	None	None

This operation is performed by the send function of the Comm module.

3. Finally the parallel vector of functions is produced. Each process i holds an array a_i of size p (a column of the previous matrix) and the function is **fun** x →a_i.(x). In our example at process 3, $(f_3\ 0) =$ None which means that process 3 received no message from process 0 and $(f_3\ 2) =$ Some v_2 which means that process 2 sent the value v_2 to process 3.

4 Performance Prediction

One of the main advantages of the BSP model is its cost model: it is rather simple but yet accurate. Several papers (for example [10]) have demonstrated this fact using the BSPlib [8] library or the Paderborn University BSP Library (PUB) [2]. In his book [1], Bisseling presents in the first chapter the BSP model, programming using the BSPlib library and examples. Among this examples, the "probe" program is a benchmark used to determine the parameters of the BSP machines.

4.1 Benchmarking the BSP Parameters

There are four BSP parameters. Of course, the number of processors does not need to be benchmarked. The g and L parameters are expressed as multiples of the processors speed r. This parameter is the first to be determined. It is also the most difficult to determine accurately.

In [1], the target applications belong to the scientific computing area. Thus r is obtained by timing several computations using floating point operations. The first version of our bsmlprobe program does almost the same, because we would like to perform some comparisons with the C version (we use an average of the measured speeds, called r and the best of the speeds, called r'). But of course, arrays are not the most commonly used data structure in Caml. For applications with rich data structures, this value of r may be not very accurate. This way to obtain r is also more favorable to C as more complex data structures, or even function calls are more efficient in Objective Caml than in C. Thus in practice, for a general usage, the way to determine r should not only rely on floating arithmetics.

Then g and L are determined as follows: several super-steps are timed, with an increasing h relation. In these super-steps, each process sends to each other process n/p or $(n/p)+1$ words. Then the least squares method is used to obtain g and L.

The average results obtained by running the probe (C+MPI) and bsmlprobe (BSML 0.3 with MPI implementation of the Comm module) programs 10 times on a 6 Pentium IV nodes cluster interconnected with a Gigabit Ethernet network are as follows:

- **C+MPI:** $r = 478$ Mflops, $g = 25.2$ Flops/word, $L = 623141$ Flops
- **BSML(MPI):** $r = 469$ Mflops, $g = 28$ Flops/word, $L = 227512$ Flops

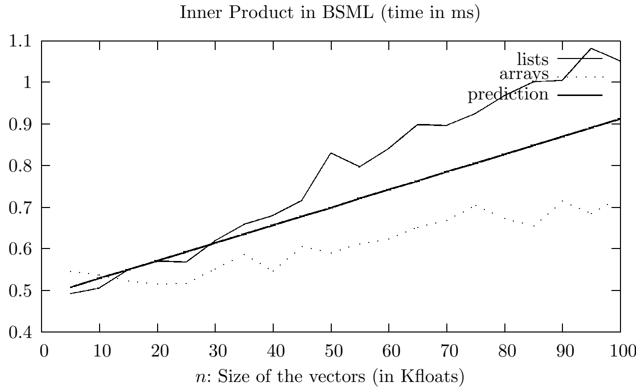
4.2 Experiments

We then performed some experiments on the program which compute the inner product of two vectors. There are two versions, one using arrays to store the vectors and one using lists to store them:

```
let inprod_array v1 v2 = let s = ref 0. in
    for i = 0 to (Array.length v1)−1 do s:=!s+.(v1.(i)*.v2.(i)); done; !s
let inprod_list v1 v2 = List.fold_left2 (fun s x y→s+.x*.y) 0. v1 v2
let inprod seqinprod v1 v2 = let local_inprod = parfun2 seqinprod v1 v2 in
  Bsmlcomm.fold_direct (+.) local_inprod
```

This code uses the fold_direct function from the BSML standard library (presented in section 2.2). Thus the overall BSP cost formula of inprod is: $n + 2 \times p + (p-1) \times g + L$.

These programs were run 15 times each for increasing size of vectors (from 5.000 to 100.000) and the average was taken. The following figure summarizes the timings. The predicted performances using the parameters obtained in the previous section are also given:



5 Related Work

The first libraries devoted to BSP programming were libraries for C which converged to a standard proposal and the Oxford BSPlib [8]. Being a library for C, it is of course very different from BSML. First the programming style is SPMD and the communication function bsp_put in *Direct Remote Memory Access* (DRMA) style may cause indeterminism if two processors try to write into the same memory zone of a third processor. The synchronization barrier should be explicitly called with bsp_sync. There are no updates of this library since 1998. The *Paderborn University BSPlib* (PUB) [2] library is also dedicated to BSP programming. It offers additional features not part of the standard. For example it is possible to have threads and the library takes care of avoiding the mismatch of messages.

There are other libraries for *Coarse-Grained Multicomputer* (CGM) programming. CGM is a special case of the BSP model and some of these libraries allow to program BSP algorithms which are not CGM algorithms [3]. NestStep [11] is an extension of C, C++ and Java for SPMD programming of BSP algorithms. It is based on a virtual shared memory on top of message passing libraries.

To our knowledge, two algorithmic skeletons languages are based on the BSP model [16, 4]. They have of course the advantages and drawbacks of skeleton approaches. Implementations have not been released. VEC-BSP has the advantage of having a cost algebra associated with the language which allows automatic performance prediction.

BSP-Haskell [13] and BSP Python [9] are inspired from the BSλ-calculus and BSML. BSP Python adapts BSML to Python which is an object oriented language. Thus elements of BSML are present. There are parallel vectors which are instances of a global class which has a method **put** for communications, *etc.*. The constructor of the ParData class takes as argument a function like BSML **mkpar**. BSP-Haskell relies on monads and it allows to avoid the evaluation of nested parallel vectors.

6 Conclusions

The BSML library allows declarative parallel programming in a safe environnement. Being implemented using Objective Caml and using a modular approach which allows to perform the communications with various communication libraries, it is portable, and efficient, on a wide range of architectures. The basic parallel operations of BSML are Bulk Synchronous Parallel operations, thus allow accurate and portable performance prediction.

Acknowledgements

This work is supported by the ACI Grid program from the French Ministry of Research, under the project Caraml (www.caraml.org).

References

1. R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.
2. O. Bonorden, B. Juurlink, I. von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.
3. A. Chan and F. Dehne. CGMgraph/CGMlib: Implementing and Testing CGM Graph Algorithms on PC Clusters. In *Proceedings of the 10th EuroPVM/MPI conference*. Springer, 2003.
4. M. Cole and Y. Hayashi. Static Performance Prediction of Skeletal Programs. *Parallel Algorithms and Applications*, 17(1):59–84, 2002.
5. F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):365–376, 2003.
6. F. Gava and F. Loulergue. A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting. *Future Generation Computer Systems*, 2005. to appear.
7. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation Series. MIT Press, 1994.
8. J.M.D. Hill, W.F. McColl, and al. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.
9. K. Hinsen. High-Level Parallel Software Development with Python and BSP. *Parallel Processing Letters*, 13(3):461–472, 2003.
10. S.A. Jarvis, J.M.D Hill, C.J. Siniolakis, and V.P. Vasilev. Portable and architecture independent parallel performance tuning using BSP. *Parallel Computing*, 28:1587–1609, 2002.
11. Christoph Kessler. Managing distributed shared arrays in a bulk-synchronous parallel environment. *Concurrency and Computation: Practice and Experience*, 16:133–153, 2004.
12. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System release 3.08, 2004. web pages at www.ocaml.org.
13. Q. Miller. BSP in a Lazy Functional Context. In *Trends in Functional Programming*, volume 3. Intellect Books, may 2002.

14. M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.
15. Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
16. A. Zavanella. Skeletons, BSP and performance portability. *Parallel Processing Letters*, 11(4):393–405, 2001.