Implementation of the BSMLlib Library v0.2

Frédéric Loulergue

Technical Report TR-2002-11

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
Implementation of a the BSMLlib Library v0.2

Frédéric Loulergue
Laboratory of Algorithms, Complexity and Logic
University Paris XII, Val-de-Marne
61, avenue du général de Gaulle
94010 Créteil cedex – France
Tel: +33 (0)1 45 17 16 50
Fax: +33 (0)1 45 17 66 01
loulergue@univ-paris12.fr

September 10, 2002

Abstract

The BSMLlib is a library for Bulk Synchronous Parallel (BSP) programming with the functional language Objective Caml. It is based on an extension of the λ-calcul by parallel operations on a parallel data structure named parallel vector, which is given by intention. A first implementation of this library was based on the BSPlib library, which is not longer supported nor updated. Being the basis of a framework for Grid computing, a new implementation of the BSMLlib based on MPI has been designed. Experimental results on a cluster of PCs are presented.

Keywords: Parallel Programming, Bulk Synchronous Parallelism, Functional Programming, MPI

1 Introduction

Some problems require performance that only massively parallel computers offer whose programming is still difficult. Works on functional programming and parallelism can be divided in two categories: explicit parallel extensions of functional languages — where languages are either non-deterministic [16] or non-functional [2, 5] — and parallel implementations with functional semantics [1] — where resulting languages do not express parallel algorithms
directly and do not allow the prediction of execution times. Algorithmic
skeleton languages [4, 17], in which only a finite set of operations (the skele-
tons) are parallel, constitute an intermediate approach. Their functional
semantics is explicit but their parallel operational semantics is implicit. The
set of algorithmic skeletons has to be as complete as possible but it is often
dependent on the domain of application.

The design of parallel programming languages is therefore a tradeoff be-
tween:

- the possibility of expressing parallel features necessary for predictable
efficiency, but which make programs more difficult to write, to prove
and to port

- the abstraction of such features that are necessary to make parallel pro-
gramming easier, but which must not hinder efficiency and performance
prediction.

We are exploring thoroughly the intermediate position of the paradigm
of algorithmic skeletons in order to obtain universal parallel languages where
execution cost can be easily determined from the source code (in this context,
cost means the estimate of parallel execution time). This last requirement
forces the use of explicit processes corresponding to the parallel machine’s
processors. Bulk Synchronous Parallel (BSP) computing [14] is a parallel
programming model which uses explicit processes, offers a high degree of
abstraction and yet allows portable and predictable performance on a wide
variety of architectures.

A denotational approach led us to study the expressiveness of functional
parallel languages with explicit processes [8] but this is not easily applicable
to BSP algorithms. An operational approach has led to a BSP λ-calculus
that is confluent and universal for BSP algorithms [13], and to a library of
bulk synchronous primitives for the Objective Caml [10] language which is
sufficiently expressive and allows the prediction of execution times [7].

This framework is a good tradeoff for parallel programming because:

- we defined a confluent calculus so

  - we can design purely functional parallel languages from it. With-
out side-effects, programs are easier to prove, and to re-use (the
semantics is compositional)

  - we can choose any evaluation strategy for the language. An eager
language will allow good performances.
• this calculus is based on BSP operations, so programs are easy to port, their costs can be predicted and are also portable because they are parametrized by the BSP parameters of the target architecture.

The version 0.1 of our BSMLlib library implements the BSML primitives using Objective Caml [10] and BSPlib [9] and its performance follows curves predicted by the BSP cost model [3]. This environment is a safe one. Our language is deterministic, is based on a parallel abstract machine [15] which has been proved correct w.r.t. the confluent B$\lambda_p$-calculus [11] using an intermediate semantics [12]. A polymorphic type system [6] has been designed, for which type inference is possible. The small number of basic operations allows BSMLlib to be taught to BSc. students.

The BSPlib library is no longer supported nor updated. Moreover BSMLlib is used as the basis for the CARAML project which aims to use Objective Caml for Grid computing with, for example, applications to parallel databases and molecular simulation. In such a context, the parallel machine is no longer a homogeneous machine as prescribe by the BSP model and global synchronisation barriers are too costly. Thus we will need encapsulated communications between different architectures and subset synchronization [18]. The new version 0.2 of the BSMLlib library is hence based on MPI [19].

This paper presents the BSP model (section 2) and gives an overview of the core BSMLlib library for functional BSP programming (section 3). We then describe the implementation of the BSMLlib library (section 4). The next section is about experiments on a cluster of PCs. We end with conclusions and future work (section 6).

2 The Bulk Synchronous Parallel Model

*Bulk-Synchronous Parallel* (BSP) computing is a parallel programming model introduced by Valiant [20] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP computer contains a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. Its performance is characterized by 3 parameters expressed as multiples of the local processing speed: the number of processor-memory pairs $p$, the time $l$ required for a global synchronization and the time $g$ for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an $h$-relation in time $gh$ for any arity $h$. 
A BSP program is executed as a sequence of supersteps, each one divided into (at most) three successive and logically disjoint phases. In the first phase each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes. In the second phase the network delivers the requested data transfers and in the third phase a global synchronization barrier occurs, making the transferred data available for the next superstep. The execution time of a superstep \( s \) is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

\[
\text{Time}(s) = \max_{i: \text{processor}} w_i^{(s)} + \max_{i: \text{processor}} h_i^{(s)} * g + l
\]

where \( w_i^{(s)} \) is the local processing time on processor \( i \) during superstep \( s \) and \( h_i^{(s)} = \max\{h_i^{(s)}, h_i^{-1}\} \) where \( h_i^{(s)} \) (resp. \( h_i^{-1} \)) is the number of words transmitted (resp. received) by processor \( i \) during superstep \( s \). The execution time \( \sum_s \text{Time}(s) \) of a BSP program composed of \( S \) supersteps is therefore a sum of 3 terms: \( W + H * g + S * l \) where \( W = \sum_s \max_i w_i^{(s)} \) and \( H = \sum_s \max_i h_i^{(s)} \).

In general \( W, H \) and \( S \) are functions of \( p \) and of the size of data \( n \), or (as in the present application) of more complex parameters like data skew and histogram sizes. To minimize execution time the BSP algorithm design must jointly minimize the number \( S \) of supersteps and the total volume \( h \) (resp. \( W \)) and imbalance \( h^{(s)} \) (resp. \( W^{(s)} \)) of communication (resp. local computation).

3 The BSMLlib Library

**BSMLlib** is based on the following elements. It is without the \( \text{pid} \) variable of SPMD programs, but uses an externally-bound variable \( \text{bsp\_p:unit\_\rightarrow int} \) such that the value of \( \text{bsp\_p()} \) is \( p \), the static number of processes. The value of this variable does not change during execution. There is also a polymorphic type constructor \( \text{par} \) such that 'a \( \text{par} \) represents the type of \( p \)-wide vectors of objects of type 'a, one per process. The nesting of \( \text{par} \) types is prohibited. A polymorphic type system enforces this restriction [6]. This improves on the earlier design DPML/Cam\l{} FLight [5] in which the global parallel control structure \( \text{sync} \) had to be prevented dynamically from nesting.

Parallel objects are created by

\[
\text{mkpar: (int \rightarrow 'a) \rightarrow 'a par}
\]

so that \( \text{mkpar f} \) stores \( (f\ i) \) on process \( i \) for \( i = 0, 1, \ldots, (p - 1) \).
A BSP algorithm is expressed as a combination of asynchronous local computations and phases of global communication with global synchronization. Readers familiar with BSMLib will observe that we ignore the distinction between a communication request and its realization at the barrier. Asynchronous phases are programmed with

\[
\text{apply: } ('a \to 'b) \text{ par} \to 'a \text{ par} \to 'b \text{ par}
\]

whose semantics is that of a map over the parallel structures. In other words \(\text{apply (mkpar } f\) (mkpar \(e\))\) stores \((f \; i)\) (\(e \; i\)) on process \(i\). Neither the implementation of BSMLib, nor its semantics [12] prescribe a synchronization barrier between two successive uses of \text{apply}.

The communication and synchronization phases are expressed by

\[
\text{put: } (\text{int} \to 'a \text{ option}) \text{ par} \to (\text{int} \to 'a \text{ option}) \text{ par}
\]

where \('a \text{ option} \) is defined by: \(\text{type } 'a \text{ option} = \text{None} | \text{Some of } 'a\).

Consider the expression:

\[
\text{put(mkpar(fun } i \to fs_i))
\]

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 \text{Some } v. To send no value from process \(j\) to process \(i\), \((fs_j \; i)\) must evaluate to \text{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 \text{None} if process \(j\) sent no message to process \(i\) or evaluates to \text{Some } v if process \(j\) sent the value \(v\) to the process \(i\).

The full language would also contain a synchronous conditional operation

\[
\text{ifat: } (\text{bool } \text{par}) \times \text{int} \times 'a \times 'a \to 'a
\]

such that \(\text{ifat } (v,i,v1,v2)\) will evaluate to \(v1\) or \(v2\) depending on the value of \(v\) at process \(i\). But Objective Caml is an eager language and this synchronous conditional operation cannot be defined as a function. That is why the core BSMLib contains the function: \(\text{at:bool } \text{par} \to \text{int} \to \text{bool}\) to be used only in the contraction: \(\text{if } (\text{at vec pid}) \text{ then... else...}\)

where \((\text{vec:boolean } \text{par})\) and \((\text{pid:int})\).

The meaning of \(\text{if } (\text{at vec pid}) \text{ then expr1 else expr2 }\) is the meaning of \(\text{ifat(vec, pid, expr1, expr2)}\).
4 Implementation of the BSMLlib Library

There are two versions of the BSMLlib library: a sequential version and a parallel version. The BS\(\lambda\)-calculus is confluent so the BSMLlib library is deterministic and the evaluation of a pure functional parallel program will lead to the same value with both versions.

In the sequential version, the abstract type 'a par is implemented as Objective Caml arrays: \texttt{type 'a par = 'a array}. In the parallel version, our operations are implemented as SPMD programs. A parallel vector is supposed to contain a value per process, so in SPMD style of programming, the abstract type 'a par is defined as \texttt{type 'a par = 'a}.

The non-communicating operations are thus very simple. The implementation is:

\begin{verbatim}
let mkpar f = f (comm_rank comm_world)
let apply f v = f v
\end{verbatim}

where \texttt{comm_rank: communicator \rightarrow int} is the Objective Caml version of the \texttt{MPI_Comm_rank} function, and \texttt{comm_world: communicator} the Objective Caml version of the communicator \texttt{MPI_COMM_WORLD}.

The implementation of the \texttt{put} operation relies on the \texttt{Marshal} module of Objective Caml (for serialization) and on the \texttt{MPI_Alltoall} functions of MPI. First, each process serializes the \(p - 1\) messages (possibly empty) to be sent to other processes. Then the sizes of the messages are exchanged between processes using the \texttt{MPI_Alltoall} function. Buffers are then allocated by each process in order to receive the messages. Messages are sent using the \texttt{MPI_Alltoallv} function. Once received, they are deserialized.

The \texttt{at} operation performs a broadcast using also one \texttt{MPI_Alltoall} in order to preserve the BSP execution model.

5 Experiments

We experimented our new implementation on a cluster of Pentium III (1 GHz, 128 Mo) with a fast ethernet network. The test programs were implementations of a BSP algorithm for reduction which:

1. performs a local reduction on each processor
2. totally exchanges the partial results
3. reduces the partial results on each processor.
The result of reduction of the whole collection is then present on each processor. We have tested four versions: reduction of an array implemented in C+MPI (using the BSP model); reduction of an array implemented with the BSMLlib library; reduction of a list implemented in C+MPI (using the BSP model); reduction of a list implemented with the BSMLlib library. The reduction was performed 100 times for each test.

The timings for arrays are given in figure 1. The timings for lists are given in figure 2. In order to compare communication efficiency, we then compared a total exchange in C+MPI and BSMLlib (fig. 3).

![C+MPI: array reduction](image1)

![BSMLlib: array reduction](image2)

Figure 1: Array Reduction
For arrays, the C+MPI version is 2.32 faster in average than the BSMLlib version. It is about 3 times better than the previous version of BSMLlib. Furthermore for lists the BSMLlib version is faster (2.46 times in average) than the C+MPI version. The step 3 of the reduction algorithm uses a list of partial results in both case. In the C version this list is obtained from the buffer obtained after the total exchange. It seems than this tranformation from an array to a list is quite expensive. Moreover, the Objective Caml compiler inlines a lot. The overhead introduced by the BSMLlib for communications is very low. In average, it is about 17% more expensive.
For a high level language, the BSMLlib library is thus very efficient. We plan to use this library for parallel symbolic applications, in which dynamic and complex data structures are used. The results for lists make us hope that we will obtain good performances.

6 Conclusions and Future Work

The BSMLlib library allows declarative parallel programming in a safe environment. Being implemented using MPI and Objective Caml, it is portable
on a wide range of architectures. The basic parallel operations of BSMLlib are Bulk Synchronous Parallel operations, thus allow accurate and portable performance prediction. The current implementation is very efficient. For lists is even outperforms a C+MPI version of the same program.

A type system has been designed and we are currently working on a type inference algorithm and its implementation. We also plan to implement an TCP/IP version of the BSMLlib library and to improve the standard library which contains commonly used parallel functions: broadcasts, total exchange, folds, scans, sorting functions, etc. Tests will be run on other clusters and on a Cray T3E.

**Acknowledgements**

This work is supported by the AC1 Grid program from the French Ministry of Research, under the project CARAML.

The author thanks the faculty of Law of the University of Paris XII for the access to the cluster of PCs used for the experiments.

**References**


Denver, Colorado, April 1995. Lawrence Livermore National Laboratory, USA.


[16] P. Panangaden and J. Reppy. The essence of concurrent ML. In F. Niel-
son, editor, *ML with Concurrency*, Monographs in Computer Science.
Springer, 1996.

Francis, 1998.

[18] D. B. Skillicorn. Multiprogramming BSP programs. Department of
Computing and Information Science, Queen’s University, Kingston,
Canada, October 1996.


[20] Leslie G Valiant. A bridging model for parallel computation. *Commu-