Efficiency of Bulk Synchronous Parallel Programming using C++, BSCF++ and BSMLlib

Dabrowski, F. and Louergue, F.

Technical Report TR-2002-10
Efficiency of Bulk Synchronous Parallel Programming using C++, BSFC++ and BSMLlib

Frédéric Dabrowski and 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
dabrowski.f@wanadoo.fr
loulergue@univ-paris12.fr

Technical Report TR2002-10

Abstract

The BSMLlib library is a library for Bulk Synchronous Parallel (BSP) 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, which is given by intention. The BSFC++ library is a library for Functional Bulk Synchronous Parallel programming in C++ which is based on the FC++ library. We present those libraries and give experimental results. For comparison, MPI/C++ versions of the same programs were also ran on our cluster of PCs.

1 Introduction

Some problems require performance that only massively parallel computers offer. But their programming is still difficult and declarative languages are needed. Works on functional programming and parallelism can be divided in
two categories: explicit parallel extensions of functional languages — where languages are either non-deterministic [17] 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, 18], in which only a finite set of operations (the skeletons) 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 between:

- 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 programming 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 [16] 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.

An operational approach has led to a BSP $\lambda$-calculus that is confluent and universal for BSP algorithms [13], and to a library of bulk synchronous primitives for the Objective Caml [10] language (called BSMLlib) which is sufficiently expressive and allows the prediction of execution times [3].

This framework is a good tradeoff for parallel programming because:

- 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.

- we defined a confluent calculus so
we can design purely functional parallel languages from it. Without 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.

In order to offer this framework to a wider audience, we have developped a C++ adaptation of our ideas. The result, called the BSFC++ library is based on the FC++ library [14, 15] and provides about the same features than BSMLlib.

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 BSFC++ library for functional bulk synchronous parallel programming in C++ (section 4). The next section is about experiments on a cluster of PCs: programs written in BSMLlib, BSFC++, but also using the MPI library (Message Passing Interface [19]) with C++ (in imperative style) were timed and the results are compared. 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 $q$ 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 (Fig. 1). 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:

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

where \(w_i^{(s)}\) = local processing time on processor \(i\) during superstep \(s\) and \(h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}\) where \(h_{i+}^{(s)}\) (resp. \(h_{i-}^{(s)}\)) 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: \(W + H \times g + S \times l\) 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

\textbf{BSMLlib} is based on the following elements. It is without the pid (for process identifier) variable of SPMD (Single Program Multiple Data) programs which makes the programs difficult to read and understand, but uses an externally-bound variable \texttt{bsp\_p\_unit\_\rightarrow\_int} such that the value of \texttt{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 \texttt{par}
such that 'a par represents the type of p-wide vectors of objects of type 'a, one per process. The nesting of par types is prohibited. A polymorphic type system enforces this restriction [6]. This improves on the earlier design DPML/Caml Flight [5] in which the global parallel control structure sync had to be prevented dynamically from nesting.

Parallel objects are created by

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

so that (mkpar f) stores (f i) on process i for i = 0, 1, ..., (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 BSP/\text{lib} will observe that we ignore the distinction between a communication request and its realization at the barrier. Asynchronous phases are programmed with

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

whose semantics is that of a map over the parallel structures. In other words apply (mkpar f) (mkpar e) stores (f i) (e i) on process i. Neither the implementation of BS\text{M}lib, nor its semantics [11] prescribe a synchronization barrier between two successive uses of apply.

The communication and synchronization phases are expressed by

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

where 'a option is defined by: type 'a option = None|Some of 'a.

Consider the expression:

\[
\text{put(mkpar(fun i->fs}_i))
\]

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.

Expression (1) evaluates to a parallel vector containing a function fd_i of delivered messages on every process. At process i, (fd_i j) evaluates to None if process j sent no message to process i or evaluates to Some v if process j sent the value v to the process i.

The full language would also contain a synchronous conditional operation

\[
\text{ifat: (bool par) * int * 'a * 'a -> 'a}
\]
such that 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 can not be defined as a function. That is why the core BSMLlib contains the function: at:bool par -> int -> bool to be used only in the construction: if (at vec pid) then... else... where (vec:boolean par) and (pid:int).

The meaning of if (at vec pid) then expr1 else expr2 is the meaning of ifat(vec,pid,expr1,expr2).

The current implementation of BSMLlib (version 0.2)\textsuperscript{1} is based on Objective Caml and MPI. It offers a standard library with common BSP algorithms (broadcasts, scatters, folds, scans, sorts, ...).

4 The BSFC++ library

BSFC++ is a C++ version of the BSMLlib using the FC++ library [15] (developed by Brian MacNamara and Yannis Smaragdakis from the Georgia institute of technologie) for functional support and MPI for basic communication operations. BSFC++ allows programming according to the BSP model except that we don't make distinction between communication operations and their synchronization barriers. Curryfication and rank 2 polymorphism are available through the use of the FC++ library.

BSFC++ programs uses an abstract type template Par < A > wich represents parallel values (one for each process). The Par class contains a member ElementType wich is used by functions to know the type of elements. As in the BSMLlib, BSFC++ does not use an externally bound variable for process identifier but there is a bsp:Par function wich returns the static number of processes and the nesting of type Par is not allowed but this responsability is let to the programmer in the current version.

4.1 The FC++ library

Before we present BSFC++ operations we have to introduce FC++ programming. C++ allows polymorphism through the use of the template mechanism but a polymorphic function cannot be passed as argument of another one. FC++ curbs this problem by the means of a new function type called functoid. A functoid is a instance of a structure whom operator() is used to defined the function. A members Sig gives the real signature of the function. Indeed the C++ type of the structure does not reflect the function type.

\textsuperscript{1}Soon available at http://www.univ-paris12.fr/laci/loulengu

6
In the following first example, the C++ type of the function is `Twice`, this doesn’t match its functional signature \texttt{int} \to \texttt{int}:

```cpp
struct Twice{
  int operator()(int x){
    return 2*x;
  }
}
```

In the second exemple the structure \texttt{Sig} which is a member of the class \texttt{CFun1Type < int, int >} represents the real functional type of the function:

```cpp
struct Twice : public CFun1Type<int,int>{
  int operator()(int x){
    return 2*x;
  }
} twice;
```

FC\++ introduces three kinds of functoids:

1. direct monomorphic functoids

2. indirect functoids wich are dynamically bound (their value can be changed). Indirect functoids permit to define variables that range over all functions with the same type signature. Indirects functoids are monomorphic.

3. direct polymorphic functoids wich use templates mechanism to allow polymorphism.

Usual C++ functions can be converted to direct monomorphics functoids by the use of the operator \texttt{ptr\_to\_fun}. Direct functoids can be converted to indirect functoids (polymorphics functoids must be monomorphized, this can be done implicitly):

```cpp
int sqrt(int x){return x*x;} // Direct polymorphic functoid
struct Sqrt{
  template <class A>
  A operator()(A x){ return x*x;}
```
template <class A>
struct Sig : public CFunType<A,A>{};

int main(){
  // Conversion from usual C++ function to direct
  // monomorphic functoid and then to indirect functoid
  Fun1<int,int> f = makefun(ptr_to_fun(&sqrt));
  // Implicit conversion from direct functoid
  // to indirect functoid
  Fun1<int,int> g = ptr_to_fun(&sqrt);
}

Notations:
1. The signature of a function is noted FunType < A_1, A_2, ..., A_n, A_R >, where A_i is the type of i^{th} argument and R is the result type.
2. RT < A, B >::ResultType is used by FC++ to get the result type of a function of type A applied to an argument of type B.

4.2 BSFC++ operations

BSFC++ operations are similar to BSMLlib operations. One can creates a parallel vector using the function:

<table>
<thead>
<tr>
<th>mkpar: template &lt;class A&gt; FunType&lt;Fun1&lt;int,A&gt;,Par&lt;A&gt;&gt;</th>
</tr>
</thead>
</table>

A parallel vector of functions can be applied to a parallel vector of values by:

| applypar: template <class A,class B> 
  FunType<Par<A>,Par<B>,Par<RT<A,B>::ResultType>> |
|--------------------------------------------------|

Messages are exchanged by the use of

| put: template<class A> 
  FunType<Par<Fun1<int,Option<A>>,Par<Fun1<int,Option<A>>>> |
|------------------------|
The argument is a parallel vector \texttt{vf} of functions from process indices to values to be sent. The type \texttt{template <\ class A > Option} encapsulates the values to be exchanged. This allows one process to send no value to another one. When the \texttt{put} function is called:

1. at each process the function held by \texttt{vf} is applied to all process indices to determine the values (if any) to be sent to other processes. Those values are copied in a buffer;

2. an array of integers containing the sizes of the values to be sent is first exchanged with other processes using the \texttt{MPI_Alltoall} function (Fig. 2). This phase is a global synchronisation barrier (a total exchange);

3. once each process knows the sizes of what it is supposed to receive, values are effectively sent using a call to \texttt{MPI_Alltoallv} (Fig. 3);

4. after values are exchanged, new functions from processes indices to values are created (one per process). For a given process, this function returns the values received by this process.

Points 1 and 4 above may need a serialization/deserialization mechanism. For this reason, \texttt{BSFC++} comes with a standard library which proposes a rudimentary mechanism of serialization in order to permit communications of complex data types such as classes and functions. If a class has to be serialized, it must inherits from the \texttt{Serializable} class.

The member methods \texttt{serialize()} and a special constructor taking a \texttt{char} buffer have to be implemented. The method \texttt{serialize} permits to the \texttt{put} function to take back the serialized value of an object, the method \texttt{pack} must be called in the \texttt{serialize} function on each members of the class. The constructor permits to build a new object by taking the buffer passed in argument, it must call unpack for each serialized value and put it in matching variables.

There is also a global parallel conditional:

\begin{verbatim}
atpar: FunType<Par<bool>,int, bool>
\end{verbatim}

The \texttt{BSFC++} library contains also a standard library similar to the standard library of the \texttt{BSMLlib} library.
MPI_Alltoall(  void *sendbuf, int sendcount, MPI_Datatype sendtype,  
void *recvbuf, int recvcount, MPI_Datatype recvtype,  
MPI_Comm comm)

- **sendbuf**: starting address of the send buffer.
- **sendcount**: number of elements in send buffer
- **sendtype**: datatype of send buffer elements.
- **recvcount**: number of elements received from any process.
- **recvtype**: datatype of receive buffer elements
- **comm**: communicator(The group of process participating to computation)

On each process $i$ the $j^{th}$ block of send buffer is sent to process $j$ and put in the $i^{th}$ block of the received buffer.

Figure 2: MPI_Alltoall

## 5 Experiments

### 5.1 Programs

We compared the performances of MPI/C++, BSFC++ and the BSMLib with two programs: a parallel reduction (of integer lists) and a total exchange of integer lists.

A total exchange is an operation such as if each processor $i$ has the value $x_i$ then after the exchange, each processor holds the values $(x_0, x_1, \ldots, x_{p-1})$. Its BSP cost is:

$$n \times p \times g + L$$

where $n$ is the size of each element $x_i$ (or the size of the biggest if they have not the same size).

For the parallel reduction, the parallel vector $(l_0, \ldots, l_{p-1})$ is considered as a whole big list $l_0@l_1\ldots@l_{p-1}$ to be reduced. The result of the reduction must be present on all the processors at the end of computation. The parallel reduction proceeds in two BSP supersteps:

1. (a) each processor performs a local reduction (asynchronous phase of the first BSP superstep)
MPI_Alltoallv( void *sendbuf, int *sendcounts, 
   int *sdispls, MPI_Datatype sendtype, 
   void *recvbuf, int *recvcounts, 
   int rdispls, MPI_Datatype recvtype, MPI_Comm comm)

- sendbuf : starting address of send buffer.
- sendcounts : integer array equal to the group size specifying the number of elements to send to each processes
- sdispls : integer array equal to the process number specifying the displacement of data destined to each processes.
- sendtype : data type of send buffer.
- recvbuf : starting address of receive buffer.
- recvcounts : integer array equal to the group size specifying the maximum number of elements which can be received from each processes.
- rdispls : integer array equal to the group sizes specifying the data displacement in recvbuf for values received from each processes.
- recvtype : datatype of receive buffer elements.
- comm : communicator.

MPI_Alltoallv realizes the same operation as MPI_Alltoall except that the messages exchanged may have variable sizes.

Figure 3: MPI_Alltoallv
(b) partial results are totally exchanged (communication and synchronization phases of the first BSP superstep)

2. reduction of the partial results on each processor (asynchronous phase of the second BSP superstep)

Its BSP cost is:

\[(n + p \times g + L) + p\]

where \(n\) is the size of each list (or the size of the biggest if they have not the same size).

For the MPI/C++ version, we use the STL (standard template library) implementation of lists whereas the BSFC++ version uses the FC++ implementation of lists.

For the total exchange operation we first make a call to \texttt{MPI\_Alltoall} to exchange lists sizes, the lists are then copied in buffers which are exchanged using the \texttt{MPI\_Alltoallv} function (Fig 3). After buffers have been exchanged, each processor builds a new list with elements from all the received lists.

For the parallel reduction we compute on each processor the sum of local lists and results are then exchanged between processors using the \texttt{MPI\_Alltoall} primitive. In order to do this we copy the result \(p\) times in a buffer and we pass it as argument of \texttt{MPI\_Alltoall} (Fig 2). Once each processor owns all the values it computes their sum.

The BSFC++ total exchange (Fig. 4) has almost the same behaviour than the MPI/C++ one except that we use the polymorphic \texttt{totex} function from the BSFC++ standard library. We create a function which returns for each processor indices the local value of the parallel vector passed as argument. This function is then passed to the \texttt{put} operation.

For the parallel reduction we use the BSFC++ operation \texttt{parallel\_reduce} (Fig. 5). This function computes the local sum of the list on each processor (line 6) and realize a \texttt{totex\_to\_list} (line 5) on the resulting parallel vector. The \texttt{totex\_to\_list} function returns the parallel vector where each value is the list of all elements of the parallel vector passed as argument. Then we compute locally the sum of resulting lists (line 4).

The \texttt{BSMLlib} programs are given in figures 6 and 7. They use the following functions from either the Objective Caml standard library or the \texttt{BSMLlib} standard library:

- \texttt{List.fold\_left}: ('a -> 'b) -> 'a -> 'b list -> 'a
- \texttt{List.map}: ('a -> 'b) -> 'a list -> 'b list
- \texttt{Bsm1base.replicate}: 'a \rightarrow 'a par defined by
  let replicate x = mkpar(fun pid \rightarrow x)

- \texttt{Bsm1base.parfun}: ('a \rightarrow 'b) \rightarrow 'a par \rightarrow 'b par defined by
  let parfun f v = apply (replicate f) v

- \texttt{Bsm1base.procs}: \texttt{unit} \rightarrow \texttt{int list}: the list of the processor identifiers.

\begin{verbatim}
1   struct Totex{
2      struct H{
3          template <class A, class B>
4              Option<A> operator()(A val, B pid) const{
5                  return Option<A>(val);
6              }
7          template <class A, class B>
8              struct Sig : public FunType<A, B, Option<A>>{};
9      }h;
10     template <class A>
11        Par<Fun1<int, Option<A>> > > operator()(Par<A> a) const{
12            return put(parfun(Curryable2<h>(h), a));
13        }
14     template <class A>
15        struct Sig : public FunType<A, Par<Fun1<int, Option<typename A::ElementType>> > > >{};
16    }totex;
\end{verbatim}

Figure 4: Total exchange: BSFC++ version

5.2 Timings

We experimented on a cluster of Pentium III (1 GHz, 128 Mo) with a fast ethernet network, with 2, 4 and 8 processors. The timings are given in 1/100 seconds.

The timings for the parallel reduction are given in figure 9. The \texttt{BSML1ib} version is about 4 times faster than the MPI/C++ version and 5 times faster than the BSFC++ version. The BSML1ib version was compiled with the native code compiler using the \texttt{-unsafe} and \texttt{-inline} options. It was impossible to obtain a correct program using \texttt{gcc} optimization options, so the MPI/C++
struct Parallel_reduce{
    template <class F, class A>
    Par<A> operator()(const F &f, const Par<List<A>> &pla) const{
        return parfun(foldl1(f),
                    to_list(parfun(Fun1<List<A>,
                                   A>(foldl1(f)), pla)));
    }
    template <class F, class A>
    struct Sig : public FunType<F, A, Par<RT<F>,
                                 typename A::ElementType::ElementType,
                                 typename A::ElementType::ElementType
                                    >::ResultType> >{
        }parallel_reduce;

Figure 5: Reduction: BSFC++ version

(* totex_list : 'a par -> 'a list par *)
let totex_list v =
    let sent = put (parfun (fun x dest->Some x) v) in
    parfun
    (List.map (fun (Some x)-> x))
    (parfun2 List.map sent (replicate (procs())))

Figure 6: Total exchange: BSMLlib version

(* reduce: ('a -> 'a -> 'a) -> 'a -> 'a list par -> 'a par *)
let reduce op v vec =
    let seq_reduce l = List.fold_left op v l in
    let local = parfun seq_reduce vec in
    parfun seq_reduce (totex_list local)

Figure 7: Reduction: BSMLlib version
Figure 8: Total exchange
and BSFC++ versions were compiled using the default optimization mode of gcc.

The timings for the total exchange are given in figure 8. For 2 and 4 processors, MPI/C++ and BSMLlib versions have similar performances and the BSFC++ version is still about 5 times slower. For 8 processors, the BSMLlib became inefficient for lists of size 4000 and 5000. We have not yet found an explanation for this behaviour.

6 Conclusions and Future Work

The BSFC++ and BSMLlib libraries allows declarative parallel programming. Being implemented using MPI and respectively C++ and Objective Caml, they are portable on a wide range of architectures. Their basic parallel operations are Bulk Synchronous Parallel operations, thus allow accurate and portable performance prediction.

The current implementation of BSMLlib is very efficient. For lists is even outperforms a MPI/C++ version of the same program. Thus declarative languages can be efficiently used for parallel programming. Of course this efficiency is obtained when using dynamic data structures like lists. For arrays, the use of the MPI library with C is more efficient [12]. The current implementation of BSFC++ is not as efficient as the BSMLlib library, but is close to the performances of the MPI/C++ version.

Nevertheless, the BSFC++ library is interesting because C++ programmers can easily use it: it requires only the MPI library and a standard C++ compiler. Skel-BSP [21] and VEC BSP[9] are two parallel skeletons languages which are compiled to MPI and C. They offer a higher-level set of parallel skeletons than BSFC++ but require dedicated compilers. Moreover their set of skeletons may only be extend at the price of an update of the whole compiler. One of our goals is to provide a new library called the BIPP library (french acronym for parallel skeleton library), which will offer a set of skeletons similar to Skel-BSP. This library is under development and is implemented using the BSFC++ library. As advocated in [7] we will not use subset synchronization (as Skel-BSP with the edD-BSP model) but usual BSP global synchronization. This makes the design of the skeletons harder but the simplier cost model will allow the design of skeletons that will be able to do dynamic load balancing which is not possible in Skel-BSP or VEC BSP.

The PML library implemented with the BSMLlib library will offer the same skeletons. This library is a subpart of the CARAML project [8] which will also contain a library for parallel databases and whose extensions will
Figure 9: Reduction
allow grid programming.

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

References


