Functional Bulk Synchronous Parallel Programming in C++

Dabrowski, F. and Loulergue, F.


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
FUNCTIONAL BULK SYNCHRONOUS PARALLEL PROGRAMMING IN C++

Frédéric Loulergue and Frédéric Dabrowski
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

Research Report TR-2002-13

Abstract

This paper presents the BSFC++ library for functional bulk synchronous parallel programming in C++. 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. This guarantees the determinism and the absence of deadlock. Broadcast algorithms are implemented using the core library.

Keywords: Parallel Programming, Functional Programming, C++, Algorithmic Skeletons

1 Introduction

Declarative parallel languages are needed to ease the programming of massively parallel architectures. Moreover, those languages does not enforce in the syntax itself an order of evaluation, and thus appear more suitable to automatic parallelization. Logic programming languages use backtracking which is very difficult to parallelize. Functional languages are more often considered. Nevertheless, even if some problems encountered in the parallelization of sequential imperative languages are avoided, some still remains (for example two different but denotationally equivalent programs may lead to very different parallel programs) and some are added, for example the fact that in those languages data-structures are always dynamic ones. It makes the amount and/or the grain of parallelism often too low.

An opposite direction of research is to give the programmer the entire control over parallelism. Message passing facilities are added to functional languages. But in this case, the obtained parallel languages are either non-deterministic [23], either non-functional (i.e. referential transparency is lost) [2, 10].

An intermediate approach is to offer only a set of algorithmic skeletons [6, 9, 25, 24, 5] (in the case of functional languages, it is a set of higher-order functions) that are implemented in parallel. Those algorithmic skeletons have sequential counterparts. For example, the map function, which takes a function f, a list \([x_0; \ldots; x_n]\) and returns the list \([(f x_0); \ldots; (f x_n)]\) is a classical algorithmic skeleton. Its usual parallel implementation scatters the list on different processors and evaluates the
(f \, x_i) \text{ in parallel, then gathers the results on one processor. The denotational semantics of the map skeletons is the same than the sequential map function. Its parallel semantics remains implicit in most algorithmic skeletons approaches.}

If parallel programming is easier using algorithmic skeletons, there are some drawbacks. Firstly, the set of algorithmic skeletons is finite and often depends on the domain of application. Most parallel languages based on algorithmic skeletons [24, 5, 31, 12, 20] rely on a specific compiler. So it is impossible for the programmer to extend himself the set of algorithmic skeletons. In other approaches [26, 8, 7, 13], the set of algorithmic skeletons is given as a library. Nevertheless those libraries are implemented using MPI [28]. Thus to program new skeletons the programmer may have to deal with indeterminism and deadlocks. Secondly, portable performance prediction is either not considered, or based on cost models too complex to be useful for the programmer (but used by the compilers).

**Bulk-Synchronous Parallel (BSP) computing** is a parallel programming model introduced by Valiant [30, 21] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. Recent works studied the BSP implementation of algorithmic skeletons [27, 31, 12]. But in most cases, the model used is extended from the BSP model by adding network's subset synchronization which is not desirable [11].

BSML [17, 3] can be seen as an algorithmic skeletons language, because only a finite set of operations are parallel, but is different by two main points:

- our operations are universal for BSP programming and thus allow the implementation of more classical algorithmic skeletons. It is also possible for the programmer to implement additional skeletons. Moreover performance prediction is possible [3] and the associated cost model is the BSP cost model. Those operations are implemented as a library for the functional programming language Objective Caml [16].

- the parallel semantics of BSML are formal ones. We have a confluent calculus [14], a distributed semantics [15] and a parallel abstract machine [22], each semantics has been proved correct with respect to the previous one.

A project on standardization of an algorithmic skeletons library began in 2001 [1]. This library will be a library for a widely used programming language (C or C++), implemented using a widely used low-level parallel programming library (MPI) and for which performance prediction will be possible and portable. Two libraries have been proposed:

- a library [7] for C and MPI where the prototypes of the algorithmic skeletons are written in the style of the prototypes of the MPI collective functions;


Our contribution aims at the design of a standard algorithmic skeletons library based on the BSP model. The first step is the design an implementation of the BSL-calculus primitives as a library for C++ and MPI. This library, called **BSFC++**, is based on the FC++ library [18, 19] for functional programming in C++.

This paper presents the BSP model (section 2) and gives an overview of FC++ library (section 3). We then describe the BSFC++ library (section 4). The next section is about examples of the use of the core BSFC++ library. 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 [30] 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 superstep](image)

A BSP program is executed as a sequence of super-steps, 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 super-step. The execution time of a super-step \( s \) is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

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

(1)

where \( w_i^{[s]} \) = local processing time on processor \( i \) during super-step \( s \) 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 super-step \( s \). The execution time \( \sum_s \text{Time}(s) \) of a BSP program composed of \( S \) super-steps is therefore a sum of 3 terms:

\[
\sum_s \text{Time}(s) = \sum_s W^{[s]} + \sum_s h_i^{[s]} * g + S * l = W + hg + Sl
\]

\( W + H \cdot g + S \cdot l \) 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 super-steps and the total volume \( h \) (resp. \( W \)) and imbalance \( h^{(s)} \) (resp. \( W^{(s)} \)) of communication (resp. local computation).
3 Functional Programming in C++

```c++
int square(int x){return x*x;} // Direct polymorphic functoid

struct Square{
  template <class A>
  struct Sig : public CFunType<A,A>{};
  template <class A>
  A operator()(A x){return x*x;}
}

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);
}
```

Figure 2: Functoids

Before we present BSFC++ operations we have to introduce FC++ programming [19, 18]. C++ allows polymorphism through the use of the template mechanism but a polymorphic function cannot be passed as argument of an other 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 define the function. A member `Sig` gives the real signature of the function. Indeed the C++ type of the structure does not reflect the function type.

In the following first example, the C++ type of the function is `Square`, this doesn’t match its functional signature `int → int`:

```
struct Square{
  int operator()(int x){
    return x*x;
  }
}
```

In the second example the structure `Sig` which is a member of the class `CFun1Type < int, int >` represents the real functional type of the function:

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

FC++ introduces three kinds of functoids:

1. direct monomorphic functoids
2. indirect functoids which 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 which use templates mechanism to allow polymorphism.

Usual C++ functions can be converted to direct monomorphic functoids by the use of the operator `ptr_to_fun`. Direct functoids can be converted to indirect functoids, polymorphic functoids must be monomorphized, this can be done implicitly (Figure 2).

**Notations:**

1. The signature of a function is noted `FunType < A_1, A_2, ..., A_n, R >`, where `A_i` is the type of the `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 Functional Bulk Synchronous Parallel Programming

**Direct broadcast:**

```cpp
struct Bcast_direct{
    template <class A>
    Par<A> operator()(int from, Par<A> value)const{
        return get_one(value, replicatepar(from));
    }
}
```

**Total-exchange broadcast:**

```cpp
struct Bcast_totex{
    template <class A>
    Par<List<A>> operator()(int from, Par<List<A>> value)const{
        return parfun(concat,totex(scatter(from,value)));
    }
}
```

Figure 3: Broadcast algorithms
Access to the architecture BSP parameters BSFC++ gives access to the underlying BSP architecture parameters. There is a `bsp_p` function which returns \( p \), the static number of processes.

`bsp_l` and `bsp_g` gives respectively the \( l \) parameter and \( g \) parameter of the BSP machine.

```
bsp_p: FunType<int>
```

Creation of parallel vectors BSFC++ programs uses an abstract type template `Par < A >` which represents parallel vector. A parallel vector contains \( p \) values (one for each process). The `Par` class contains a member `ElementType` which is used by functions to know the type of elements. The nesting of type `Par` is not allowed but this responsibility is left to the programmer.

One can creates a parallel vector using the function:

```
mkpar: template <class A>
FunType<Fun1<int,A>,Par<A>>
```

`mkpar(f)` is such that process \( i \) will hold the value \( f(i) \).

Point-to-point parallel application 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>>
```

The `mkpar` or `applypar` operation is executed in the asynchronous computations phase of a BSP super-step. Its cost is \( W = \max_{i\in\text{process}} w_i(s) \), where \( w_i(s) \) is the (sequential) time required to execute the called operation at process \( i \).

Communications 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 `vf` of functions from process indexes to values to be sent. The type template `class A > Option` encapsulates the values to be exchanged. This type has two constructors. The first one without parameter specify that no value will be send. The second one takes as parameter the value to be sent. Thus the function \( f_j \) (held by parallel vector `vf` at process \( j \)) must be such that:

1. \( f_j(i) = \text{Option} < A > (a) \) to send (from process \( j \)) the value \( a \) to process \( i \)
2. \( f_j(i) = \text{Option} < A > () \) to send no value (from process \( j \)) to process \( i \)

The returned value of function `put` is a parallel vector of functions from process indexes to received values. The function \( g_i \) held at process \( i \) by this parallel vector is such that \( g_i(j) \) is the value received at process \( i \) from process \( j \).

When the `put` function is called: (1) at each process the function held by `vf` is applied to all process indexes 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 `MPI_Alltoall` function [28]. This phase is a global synchronization
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 MPI_Alltoallv [28] ; (4) after values are exchanged, new
functions from processes indexes to values are created (one per process). For a given process, this
function returns the values received by this process.

A call to the put function is a full super-step. (1) is an asynchronous computation phase, (2)
is the synchronization barrier phase (it is why we said in section 2 the three phases of a super-
step are logically distinct and successive), (3) is the communication phase and (4) is the beginning
(asynchronous computations phase) of a second super-step. Its cost is given by general formula (1)
of section 2.

Points (1) and (4) above may need a serialization/deserialization mechanism. For this reason,
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 Serializable class.

The member methods serialize() and a special constructor taking a char buffer have to be
implemented. The method serialize permits to the put function to take back the serialized value
of an object. The method pack must be called in the serialize function on each members of the
class to be serialized. 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.

Global conditional There is also a global parallel conditional:

```
@par: FunType<Par<Bool>,Int,Bool>
```

@par can only be use in the if structure such as if (at(vb,n)) then e1 else e2 evaluates
to e1 if vb holds true at process n and to e2 if vb holds false at process n and each process
executes then the matching code. The @par function realize a simple BSP broadcast of the value
corresponding to the process index passed as argument. Its cost is \((p - 1) \times g + l\).

5 Examples

The BSFC++ library comes with a standard library (implemented with the core library presented
in the previous section). The presented broadcast algorithms (Figure 4) use some functions of the
standard library:

- \texttt{parfun}(f) transforms the sequential function \( f \) (with one parameter) to a parallel function
  which applies \( f \) at each process. \texttt{parfun2} is the same but \( f \) has 2 parameters.
- \texttt{replicatepar}(x) transforms the sequential value \( x \) to a parallel vector with the value \( x \) at
  each process.
- \texttt{get_one(values,pids)} take a parallel vector of values \( \langle v_0, \ldots, v_{p-1} \rangle \) and a parallel vector of
  process identifiers \( \langle n_0, \ldots, n_{p-1} \rangle \) and returns the parallel vector of values \( \langle x_{n_0}, x_{n_1}, \ldots, x_{n_{p-1}} \rangle \).
  It is implemented using two put operations.
- \texttt{concat} transforms a list of lists to a list. \texttt{concat} is a function of the FC++ library.
- \texttt{line 19 of Tex} in the expression \((\texttt{take(bsp.n(),enumFromTo(0,bsp.p()))})\) creates the list
  \([0;1;\ldots;(p-1)]\). \texttt{take} and \texttt{enumFromTo} are functions of the FC++ library.
```c++
struct Scatter{
  template <class A>
  Par<List<A>> operator()(int from, Par<List<A>> v) const{
    // scatters the list held at process "from" to all other processes
  }

  template <class A>
  struct Sig: public FunType<int,A,A> {};
}
scatter;

struct Totex{
  template <class A>
  struct Sig: public FunType<A,Par<List<typename A>::ElementType> > {};
}

struct H{
  template <class A, class B>
  Option<A> operator()(A val, B pid) const{
    return Option<A>(val);
  }

  template <class A, class B>
  struct Sig: public FunType<A,B,Option<A>> {};
}

h;

totex;
```

Figure 4: Scatter and total-exchange
The two programs of figure 4 are two versions of broadcast. The first one is called `Bcast_direct`. The value at process `from` is broadcast directly to other processes using `get_one`. It is implemented in two super-steps (with `put`). In the first super-step, each process requests a value to one other process. In the second super-step, the requested values are sent. The cost of this program: \((p - 1) \times g + l\) + \((s \times (p - 1) \times g + l)\) where \(s\) is the size of the value held by process `from` by the parallel vector value.

The second one is specialized for large values (in the presented algorithm, the data-structure is a list). It proceeds in two super-steps. In the first one, the value held at process `from` is scattered (cut in \(p\) pieces of about the same size, each piece is sent to one process). In the second one, the pieces are totally-exchanged (Figure 4) and then at each process, the pieces are recombined. Its cost is: \((n + (p - 1) \times s \times \frac{n}{p} \times g + l)\) + \((p - 1) \times s \times \frac{n}{p} \times g + l + n)\) where \(s\) is the size of one element of the list.

BSFC++ gives access to the BSP parameters of the machine. It is possible to write a broadcast program which chooses, by calculating the two costs, the more efficient broadcast between the direct broadcast and the total-exchange broadcast.

6 Conclusions and Future Work

The BSFC++ library allows declarative parallel programming. Being implemented using MPI and C++, it is portable on a wide range of architectures. Its basic parallel operations are Bulk Synchronous Parallel operations, thus allow accurate and portable performance prediction.

The BSFC++ library can be easily used by C++ programmers: it requires only the MPI library and a standard C++ compiler. Other algorithmic skeletons languages offer higher-level skeletons but either require dedicated compilers – in this case their set of skeletons may only be extend at the price of an update of the whole compiler – or do not offer an associated cost model and can be extended by a programmer only at the price of using MPI with the risk of non-determinism and deadlocks.

Our main goal is to provide a new library called the BiPP library (French acronym for parallel skeleton library), which will offer a classical set of skeletons. This library is under development and is implemented using the BSFC++ library. As advocated in [11] we will not use subset synchronization but usual BSP global synchronization. This makes the design of the skeletons harder but the simpler cost model will allow the design of skeletons that will be able to do dynamic load balancing in complex situations (for e.g. [4]).

Acknowledgments

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

References


