Compositionality in Functional Bulk Synchronous Parallelism

Loulergue, F.

Technical Report TR-2002-21
Compositionality in Functional Bulk Synchronous Parallelism

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 1 45 17 16 50
Fax: +33 1 45 17 66 01
loulergue@univ-paris12.fr

December 11, 2002

Abstract
The BSL\textsubscript{\lambda}-calculus an extension of the \textlambda-calculus by bulk synchronous parallel (BSP) operations on a parallel data structure named parallel vector. This paper presents how functional composition is preserved in this framework both from the semantics point of view and from the cost model point of view.

Those operations are flat and allow BSP programming in direct mode but it is impossible to express directly divide-and-conquer algorithms. This paper also present a new kind of composition, a new construction for the BSL\textsubscript{\lambda}-calculus which can express divide-and-conquer algorithms. It is called parallel superposition. An associated cost model derived from the BSP cost model is also given.

1 Introduction

Declarative parallel languages are needed to ease the programming of massively parallel architectures. Moreover, those languages do not enforce in the syntax itself an order of evaluation, and thus appear more suitable to automatic parallelization. Functional languages are often considered. Nevertheless, even if some problems encountered in the parallelization of sequential imperative languages are avoided, some still remain (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 or difficult to control in case of speculation [1, 29]. 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, [27] or non-functional (i.e. referential transparency is lost) [2, 13].

An intermediate approach is to offer only a set of algorithmic skeletons [10, 11, 28, 9, 30] (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 \texttt{map} function, which takes a function \textit{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 \((f(x_i))\) in parallel, then gathers the results on one processor. The denotational semantics of the \texttt{map} skeleton is the same as the sequential \texttt{map} function. Its parallel semantics remains implicit in most algorithmic skeletons approaches.

Among researchers interested in declarative parallel programming, there is a growing interest in execution cost models taking into account global hardware parameters like the number of processors and bandwidth. The Bulk Synchronous Parallel [35, 32, 25] execution and cost model offers such possibilities and with similar
motivations we have designed BSP extensions of the \( \lambda \)-calculus [23] and a library for the Objective Caml
language, called BSMLlib [22], implementing those extensions.

A BSP algorithm is said to be in direct mode [15] when its physical process structure is made explicit.
Such algorithms offer predictable and scalable performance and BSML expresses them with a small set of
primitives taken from the confluent BSA calculus [23]: a constructor of parallel vectors, asynchronous parallel
function application, synchronous global communications and a synchronous global conditional.

Those operations are flat: it is impossible to express directly parallel divide-and-conquer algorithms.
Nevertheless many algorithms are expressed as parallel divide-and-conquer algorithms [33] and it is difficult to
transform them into flat algorithms. In a previous work, we proposed an operation called parallel composition
[21], but is was limited to the composition of two terms whose evaluations require the same number of BSP
super-steps. In this paper we present a new operation called parallel superposition which can be used to
write divide-and-conquer algorithms and the associated cost model. The semantics and the cost model are
compositional. The presentation of those novelties needs us to show the compositional nature of the flat
BSA\(_\lambda\)-calculus and its associated cost model.

We first present the BSP model (section 2). Section 3 is about the BSA\(_f\)-calculus and functional composition
as well as the associated compositional cost model. Section 4 introduces a new operation called parallel
superposition to the BSA\(_\lambda\) calculus and the cost model associated to this extended calculus. We then discuss
related work (section 5) and conclude (section 6).

2 Bulk Synchronous Parallelism

Bulk-Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [35,
32, 25] 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:

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

where \( w_i^{(s)} \) = local processing time on processor \( i \) during superstep \( s \) and \( h_i^{(s)} = \max\{h_i^{(e)}, h_i^{(p)}\} \) where
\( h_i^{(e)} \) (resp. \( h_i^{(p)} \)) 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:

\[
    W + H + s \times S + l
\]

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 \( n_i \), or 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 \( h^{(e)} \) (resp. \( W^{(e)} \)) of communication (resp. local computation).
3 Functional Composition

3.1 The BS$\lambda_p$-calculus

In this section we introduce an extension of the $\lambda$-calculus called the BS$\lambda_p$-calculus. Its parallel data structures are flat and map directly to physical processors. This difference with certain languages, although apparently minor is crucial: BS$\lambda_p$ programs require no flattening [5] and have thus complete control of the computation/communication ratio. The calculus introduces operations for data-parallel programming but with explicit processes in the spirit of BSP. The presentation is slightly different from [20], in particular the get operation has been replaced by a more general communication operation put.

3.1.1 Syntax

We consider a set $V$ of local variables and a set $\bar{V}$ of global variables. Let $\bar{x}, \bar{y}, \ldots$ denote local variables and $\bar{z}, \bar{y}, \ldots$ denote global variables from now on. $x$ will denote a variable which can be either local or global.

The syntax of BS$\lambda_p$ begins with local terms $e$: $\lambda$-terms representing programs or values stored in a processor's local memory:

$$e ::= n \quad \text{integer constant}$$
$$\mid b \quad \text{boolean constant}$$
$$\mid \text{nc} \quad \text{"no communication" constant}$$
$$\mid \text{isnc} \quad \text{test for nc}$$
$$\mid \bar{x} \quad \text{local variable}$$
$$\mid \oplus \quad \text{usual operations on integers and booleans}$$
$$\mid \lambda \bar{x}. e \quad \text{lambda abstraction}$$
$$\mid e\ e \quad \text{application}$$
$$\mid \text{if } e \text{ then } e \text{ else } e \quad \text{conditional}$$

The set of processor names $N$ is a set of $p$ closed local terms in normal form. In the following will we consider that $N = \{0, \ldots, p-1\}$. The $\text{nc}$ constant and the test for this constant are used in conjunction with communication operations. $\text{nc}$ means "no communication" and is used to avoid the sending of a message to a processor.

The principal BS$\lambda_p$ terms $E$ are called global and represent parallel vectors i.e. tuples of $p$ local values where the $i$\textsuperscript{th} value is located at the processor with rank $i$. The notation is: $\langle e_0, \ldots, e_{p-1} \rangle$ where $e_0, \ldots, e_{p-1}$ are local terms. The width of parallel vectors is $p$ the number of processors of the BSP machine. There is one semantics per value of $p$.

The programming languages based on this framework offer syntactic construction which allow to write programs parameterized by this value $p$. In the BSML1.0b library [22] for the Objective Caml language parallel vectors are created by the intentional parallel vector constructor $\text{mklpar}$ which takes as argument a function from integers (processor names) to values. $\text{mklparf}$ evaluates to a parallel vector such as processor $i$ holds the result of the evaluation of $f i$. Of course this evaluation depends on $p$ the number of processors: the width of the parallel vector is $p$. Such a constructor can be defined in the BS$\lambda_p$ calculus (see section 3.1.3). The number of processors of the machine is also accessible in the library with the function $\text{bsp_p}$ which returns it.
The global terms are given by the following grammar\(^1\):

\[
E ::= \tilde{x} \quad \text{global variable}
\]

| \( E E \) | application to a global term
| \( E e \) | application to a local term
| \( \lambda \tilde{x}. E \) | abstraction of a global variable
| \( \lambda \tilde{x}. E \) | abstraction of a local variable
| \( (e_0, \ldots, e_{p-1}) \) | parallel vector
| \( E \# E \) | pointwise parallel application
| \( ! E \) | put
| \( (E \rightarrow E, E) \) | global synchronous conditional
| \( (e \rightarrow E, E) \) | conditional

and has the following denotational meaning.

Global terms denote parallel vectors (finite maps from \( N \) to local values) functions between them (\( \lambda \tilde{x}. E \)) or functions from local values to such vectors (\( \lambda \tilde{x}. E \)).

The forms \( E_1 \# E_2 \) and \( ! E \) are called parallel application and put respectively. \( \# \) represents pointwise application of a vector of functions to a vector of values. i.e. the pure computation phase of a BSP superstep. Put is used for communications (followed by a synchronization barrier). The exact meanings of those operations are defined by the \( BSL_p \) rules.

The last forms of global terms define synchronous and asynchronous conditional expressions. The meaning of the synchronous conditional \( (E_1 \rightarrow E_2, E_3) \) (not to be confused with \( \text{if} e_1 \text{then} E_2 \text{else} E_3 \)) is that of \( E_2 \) (resp. \( E_3 \)) if the vector denoted by \( E_1 \) has value \text{true} (resp. \text{false}) at the processor name denoted by \( n \).

Global conditional is used to express algorithms like:

\[
\text{Repeat}
\]

\[
\text{Parallel Iteration}
\]

\[
\text{Until Max of local errors < epsilon}
\]

because without them, the global control can not take into account data computed locally, i.e. global control can not depend on data.

The global synchronous conditional induces a global synchronization and takes its decision from a vector of booleans. Nevertheless, it may be useful to be able to take a global decision from a boolean. Such a conditional doesn't induce a global synchronization. All the processors compute the value of the condition. As this condition is a local term, its value doesn't depend on the processor where it is evaluated. As a consequence, all the processors have the same value and take the same decision.

In the following we will identify terms modulo renaming of bound variables and we will use Barendregt's variable convention\(^3\): if terms \( e_1, \ldots, e_p \) occur in a certain mathematical context then in these terms all bound variables are chosen to be different from free variables.

### 3.1.2 Rules

We now define the reduction of \( BSL \) terms.

The reduction of local terms is simply \( \beta \)-reduction, obtained from the local \( \beta \)-contraction rule

\[
(\lambda \tilde{x}. e)e' \rightarrow e[\tilde{x} \leftarrow e']
\]

with the usual context rules, and rules (not detailed here) for usual operations on integers and booleans, the test for the \text{nc} constant, and conditionals.

The reduction of global terms is defined by syntax-directed rules and context rules which determine the applicability of the former. There is one semantics per value of \( p \) the number of processors of the BSP

---

\(^1\)Terms of the form \( \lambda \tilde{x}. E \) (resp. \( \lambda \tilde{x}. E \rightarrow E' \)) are not \( \beta \)-contracted and constitute implicit errors because they present a local argument to a global \( \rightarrow \) global function (resp. a global argument to a local \( \rightarrow \) global function). In practice a two-level polymorphic type system\(^4\) eliminate them, and their is no syntactic difference between local and global variables. Their sort is inferred by the type system, but we will not discuss this here.
machine. First, there are rules for global beta-equivalence:

\[(\lambda \hat{x}. E) E' \rightarrow E[\hat{x} \leftarrow E']\]  
\[(\lambda \hat{x}. E) e' \rightarrow E[\hat{x} \leftarrow e']\]  

There are also axioms for the interaction of the vector constructor with the other BSP operations (figure 1). The above rules are applicable in any context:

\[
\langle t_0, \ldots, t_{p-1} \rangle \# \langle u_0, \ldots, u_{p-1} \rangle \rightarrow \langle e_0 u_0, \ldots, e_{p-1} u_{p-1} \rangle
\]
\[
\langle (t_0, \ldots, t_{p-1}) \rangle \rightarrow \langle \ldots, \lambda j. \text{if in}_N j \text{then} (e_j) \text{else} \text{nc}, \ldots \rangle
\]

where for all \(i \in \{0, \ldots, p-1\}\), \(n\) is a processor name belonging to \(N\), and \(\text{in}_N\) is a function returning \(\text{true}\) if its arguments belong to \(N\) and \(\text{false}\) otherwise.

Figure 1: Rules for global terms

The meaning of the rule (5) for put is the following. The put operation takes a parallel vector of functions \(\langle f_0, \ldots, f_{p-1} \rangle\). Each \(f_i\) is a function from integers (processor’s names) to values. If at processor \(j\), the function \(f_j\) applied to processor name \(i\) evaluates to \(\text{nc}\), there is no communication. Otherwise, the given value is sent to the processor \(i\) by processor \(j\). The resulting vector is a vector of functions \(\langle g_0, \ldots, g_{p-1} \rangle\). At processor \(i\), the function \(g_i\) applied to \(j\) evaluates to the value of the function \(f_j\) applied to \(i\) if there is a communication, \(\text{nc}\) otherwise. Note that the right hand side of this rule is not really a global term, but rather a meta-term. A real term would contain a nesting of conditionals.

The BS\(\lambda\rhd\) calculus is a confluent calculus [20].

3.1.3 Examples

\begin{align*}
\text{get} & = \lambda \hat{d}. \lambda \hat{s}. \text{reply} \& \hat{s} \\
\text{reply} & = !((\pi (\lambda i. \text{Fr}) \# \text{ask}) \# \hat{d}) \\
\text{ask} & = !((\pi (\lambda i. \text{Fa}) \# \hat{s}) \\
\text{Fr} & = \lambda f. \lambda \hat{d}. \lambda j. \text{if} (f \ j) = \text{true then} \hat{d} \text{else} \text{nc} \\
\text{Fa} & = \lambda \hat{s}. \lambda j. \text{if} \ j = \hat{s} \text{then} \text{true else} \text{nc}
\end{align*}

Figure 2: Example: the get operation

1. The first example shows that the intentionality is not lost. The \(\pi\) or parallel vector constructor of BS\(\lambda\) can be defined as: \(\pi = \lambda f. \langle f 0, \ldots, f (p-1) \rangle\). In the following examples, we will use the \(\pi\) function to write programs.

2. The second example (figure 2) shows that the get operation described in [20] can be defined using the put operation. In the first super-step, each processor requests the sending of a value forum the process
\[
\begin{align*}
\text{bcast } & 0 \langle t_0, \ldots, t_{p-1} \rangle \\
\xrightarrow{[2]} \quad \quad \text{get } & \langle t_0, \ldots, t_{p-1} \rangle \pi(\lambda i.0) \\
\xrightarrow{[3]} \quad \quad \text{get } & \langle t_0, \ldots, t_{p-1} \rangle \langle (\lambda i.0)0, \ldots, (\lambda i.0)(p-1) \rangle \\
\xrightarrow{[1]} \quad \quad \text{get } & \langle t_0, \ldots, t_{p-1} \rangle \langle 0, \ldots, 0 \rangle \\
\xrightarrow{[5]} & \langle t_0, \ldots, t_0 \rangle 
\end{align*}
\]

Figure 3: Example of reduction

given by the second argument. In the second super-step, the requested processes send their values. The get operations is such as:

\[
\text{get } \langle t_0, \ldots, t_{p-1} \rangle \langle n_0, \ldots, n_{p-1} \rangle \longrightarrow^* \langle t_{n_0}, \ldots, t_{n_{p-1}} \rangle
\]

3. The third one is the direct broadcast algorithm which broadcasts the value held at processor \( n \):

\[
bcast = \lambda \tilde{n}. \lambda \tilde{u}. \text{get } \tilde{u} \pi(\lambda i.\tilde{n})
\]

If applied to an integer and a parallel vector it can be reduced as partially shown in figure 3.

3.2 Functional Composition

We have to define several compositions\(^2\):

- one for local terms: \( \delta = \lambda \tilde{f}. \lambda \tilde{g}. \lambda \tilde{x}. \tilde{f}(\tilde{g} \tilde{x}) \)
- two for global terms:
  - \( \bar{\delta} = \lambda \tilde{f}. \lambda \tilde{g}. \lambda \tilde{x}. \tilde{f}(\tilde{g} \tilde{x}) \)
  - \( \bar{\delta} = \lambda \tilde{f}. \lambda \tilde{g}. \lambda \tilde{x}. \tilde{f}(\tilde{g} \tilde{x}) \)

Those compositions are usual functional compositions. The BS\(\lambda_p\) is a confluent calculus, thus the order of reduction can be anything and is context independent.

3.3 Cost model

A formal parallel cost model can be associated to reductions in the BS\(\lambda\)-calculus [20]. “Cost terms” are defined, and each rule of the semantics is associated to a cost rule on cost terms.

Given a particular strategy of reduction, a term is always reduced in the same way. In this case costs can be associated with terms rather that reductions. It is the way we choose in this paper to ease the discussion about the compositional nature of the cost model and the presentation of the cost model of section 4.

For the BS\(\lambda_p\)-calculus it is possible to define two different reduction strategies for the two levels of the calculus. We choose here the same strategy for local and global reduction: weak call-by-value strategy. The strategy can be roughly described as follows:

- reduction is impossible in the scope of a \( \lambda \), the last two arguments of a conditional (either synchronous or asynchronous) cannot be reduced
- for application, parallel application, put and other operations, the arguments are evaluated from right to left; for synchronous global conditionals the processor name is first evaluated then the condition; for conditionals (local and asynchronous global) the condition is first evaluated;

\(^2\)Using the two level polymorphic type system, a single definition is needed
• for application, parallel application, put and other operations, the non-context rules can be applied only if the arguments are values (i.e. abstractions, constants, parallel vector of constants); for synchronous global conditionals the rule can be applied when the terms representing the processor name and the condition are values, for conditionals the rule can be applied when the term representing the condition is a value.

No order of reduction is given between the different components of a parallel vector. Theirs evaluations are done in parallel and this appears in the semantics as any possible interleaving applications of rules for local terms. The cost is this case is independent from the order of reduction.

We will not describe a the cost of the evaluation of a local term, it is the same than a strict functional language (ML for example), but give the costs of the evaluation of global terms.

The cost model associated to ours terms follows the BSP cost model. The evaluation of a parallel vector occurs in the first phase of the BSP superstep. If the sequential evaluation time of each component of the parallel vector is \( w_i \), the parallel evaluation time of the parallel vector is \( \max_{0 \leq i < p} w_i \).

Provided the two arguments of the parallel application are values (in this case parallel vectors of values), the parallel evaluation time of \( \langle f_0, \ldots, f_{p-1} \rangle \# \langle v_0, \ldots, v_{p-1} \rangle \) is \( \max_{0 \leq i < p} w_i \) where \( w_i \) is the evaluation time of \( f_i \) \( v_i \) at processor \( i \).

The evaluation of a put operation requires a full superstep. To evaluate \( \langle f_0, \ldots, f_{p-1} \rangle \), first each processor evaluates the \( p \) local terms \( f_j \), \( 0 \leq j < p \) leading to \( p \) values (\( p \) per processor) \( v_j \) (first phase of the superstep). If the value \( v_j \) of processor \( i \) is different from \( v_j \) of processor 0 it is sent to processor \( j \) (communication phase of the superstep). Once all values have been exchanged a synchronization barrier occurs (third and last phase of the superstep) making the values available for the next superstep. At the beginning of this second superstep, each processor \( i \) constructs the function (result of the put operation) from the \( v_j \) values it has received. Thus provided the argument of the put operation is a value (in this case a parallel vector of values), the parallel evaluation time of \( \langle f_0, \ldots, f_{p-1} \rangle \) is

\[
\max_{0 \leq i < p} w_i^1 + \max_{0 \leq i < p} h_i \times g + l + \max_{0 \leq i < p} w_i^2
\]

where

• \( w_i^1 \) is the evaluation time at processor \( i \) of the \( p \) terms \( f_j \), \( 0 \leq j < p \)

• \( h_i = \max\{ h_{i+}, h_{i-} \} \) where \( h_{i+} \) (resp. \( h_{i-} \)) is the number of words transmitted (resp. received) by processor \( i \). \( h_{i+} \) is the sum of the size of the \( v_j \) values sent to other processors, and \( h_{i-} \) is the sum of the sizes of the \( v_j \) values received by processor \( i \) from other processors

• \( w_i^2 \) is the evaluation time at processor \( i \) to construct the result function from the \( v_j \) values.

The evaluation of a global conditional \( (\langle b_0, \ldots, b_{p-1} \rangle \nRightarrow E_1, E_2) \) where \( n \) and \( b_i \) are respectively integer and boolean values is: first the processor \( n \) send the value \( b_n \) to all other processors. A synchronization barrier occurs. If the value \( b_n \) is \textit{true} (resp. \textit{false}) then the evaluation of \( E_1 \) (resp. \( E_2 \)) begins. The computation fails if the condition \( 0 \leq n < p \) does not hold. The parallel evaluation time is \( (p-1) \times s_{\text{boolean}} \times g + l + T \) where \( s_{\text{boolean}} \) is the size in words of a boolean value and \( T \) the parallel evaluation time of \( E_1 \) (resp. \( E_2 \)).

The costs (parallel evaluation times) above are context independent. The time required to evaluate a global term \( E \) will be the same in \( (\lambda x. E') E \), \((E) E \# E' \), etc. This is why our cost model is \textit{compositional}.

The compositional nature of this cost model relies on the absence of nesting of parallel expressions. In this paper this nesting is enforced by the syntax (a parallel vector can only contain local terms but no global terms) and the \( \beta \)-reduction rules (in order to avoid the reduction of terms like \( (\lambda x. E') E \) which would replace a local variable by a global term). If the nesting was not forbidden, for a parallel vector \( v \), the \textit{bcast} function defined in section 3.1.3, the expression \( \text{ex}_i = \pi(\lambda i. \text{bcast} \ i \ v) \) would be a correct one. A first problem is the meaning of this expression. We said that \( \pi f \) evaluates to a parallel vector such that process \( i \) holds value \( (f_i) \). In the case of our example, it means that process 0 should hold the value of \( \text{bcast} \ 0 \ v \). Since the \( \text{BSA}_p \)-calculus is confluent, it is possible to evaluate \( \text{bcast} \ 0 \ v \) sequentially. But in this case the execution time will not follow the formula given by the above cost model. The cost of an expression will then depend on its context. The cost model will no more be compositional.
We could also choose that process 0 broadcasts the expression (\texttt{bcast 0 v}) and that all processes evaluate it. In this case the execution time will follow the above cost model. But the broadcast of the expression will need communications and synchronization. This preliminary broadcast is not needed if (\texttt{bcast 0 v}) is not under a \texttt{?}. Thus we have additional costs that make the cost model still non-compositional. Furthermore, this solution would imply the use of a scheduler and would make the cost formulas very difficult to write.

4 Parallel Superposition

4.1 Syntax and Semantics

In order to express parallel superposition, we need to add pairs and projections of global terms. The parallel superposition is noted \( E_1 \Join E_2 \) and informally it means that global term \( E_1 \) will be reduced concurrently to \( E_2 \), one (parallel) thread to reduce \( E_1 \) and another to reduce \( E_2 \).

\[
E ::= \begin{array}{ll}
\text{parallel superposition} & E \Join E \\
\text{pair of global terms} & (E, E) \\
\text{first projection} & \text{Fst } E \\
\text{second projection} & \text{Snd } E
\end{array}
\]

At the same level of description than the previous section, parallel superposition is the same than pairing as shown by rule (11). The two additional rules are for projections:

\[
\begin{align*}
E_1 \Join E_2 & \rightarrow (E_1, E_2) \\
\text{Fst } (E_1, E_2) & \rightarrow E_1 \\
\text{Snd } (E_1, E_2) & \rightarrow E_2
\end{align*}
\]

4.2 Cost Model

To evaluate a parallel superposition, \( E_1 \Join E_2 \) the parallel machine will use a thread to evaluate \( E_1 \) and a thread to evaluate \( E_2 \). Nevertheless those two threads will not be independent. The communications and synchronization barrier will be shared.

Let consider the beginning of the evaluation of \( E_1 \Join E_2 \). First the asynchronous computation phases of \( E_1 \) and \( E_2 \) will be evaluated. This can be done using a thread mechanism but it also be done sequential to avoid thread swapping overhead. Then the communications phases of \( E_1 \) and \( E_2 \) will be done together. The messages will simply be the concatenations of the messages of \( E_1 \) and \( E_2 \). Finally a single synchronization barrier will occur to end the super-step of \( E_1 \) and the super-step of \( E_2 \).

To determine the cost of the evaluation of \( E_1 \Join E_2 \) it is not sufficient to consider the total costs of \( E_1 \) and \( E_2 \) (in the form \( W + H \times g + L \)) but the list of the costs of each super-step of \( E_1 \) and \( E_2 \). Moreover the cost of a super-step have to be described by three vectors :

- the cost of the local computations times for each process: \( \langle w_0, \ldots, w_{p-1} \rangle \),
- the size of the sent messages : \( \langle h^+_0, \ldots, h^+_p \rangle \)
- the size of the received messages : \( \langle h^-_0, \ldots, h^-_p \rangle \)

We will note \((\vec{w}, \vec{h}^+, \vec{h}^-)\) the cost of a super-step.

If the costs of \( E_1 \) and \( E_1 \) are respectively :

\[
(\vec{w}^0, \vec{h}^{0+}, \vec{h}^{0-}), \ldots, (\vec{w}^{k_1}, \vec{h}^{k_1+}, \vec{h}^{k_1-})
\]

and

\[
(\vec{w}^0, \vec{h}^{0+}, \vec{h}^{0-}), \ldots, (\vec{w}^{k_2}, \vec{h}^{k_2+}, \vec{h}^{k_2-})
\]

then the cost of \( E_1 \Join E_2 \) is
\[
\left( \begin{array}{c}
\langle w_0^0 + w_0^0, \ldots, w_0^0 + w_0^0 \rangle \\
\langle h_0^0 + h_0^0, \ldots, h_0^0 + h_0^0 \rangle \\
\langle h_0^0 + h_0^0, \ldots, h_0^0 + h_0^0 \rangle \\
\end{array} \right), \ldots, \left( \begin{array}{c}
\langle w_p^k + w_p^k, \ldots, w_p^k + w_p^k \rangle \\
\langle h_p^k + h_p^k, \ldots, h_p^k + h_p^k \rangle \\
\langle h_p^k + h_p^k, \ldots, h_p^k + h_p^k \rangle \\
\end{array} \right)
\]

where \( k = \max\{k_1, k_2\} \) and where \( w_i^n, h_i^n \) and \( h_i^n \) (resp. \( w_i^n, h_i^n \) and \( h_i^n \)) are considered equal to 0 if \( n > k_1 \) (resp. \( n > k_2 \)).

The usual BSP cost of \( E_1 \cup E_2 \) is then

\[
\left( \sum_{n=0}^{k} \max\{W^n\} \right) + \left( \sum_{n=0}^{k} h^n \right) + k \times l
\]

where \[
\begin{align*}
W^n &= \max_{0 \leq i < p} \{w_i^n + w_i^n\} \\
h^n &= \max_{0 \leq i < p} \{h_i^n + h_i^n\}, \ (h_i^n + h_i^n)
\end{align*}
\]

Using the above lists, the compositionality of our cost model is preserved.

The parallel superposition of \( E_1 \) and \( E_2 \) may be less costly than the evaluation of \( E_1 \) followed by the evaluation of \( E_2 \).

### 4.3 Programming with the parallel superposition

Objective Caml is an eager language. To express parallel superposition as a function we have to “freeze” the evaluation of its parallel arguments. So parallel superposition must have the following type:

```
super: (unit -> 'a par) -> (unit -> 'b par) -> 'a par * 'b par
```

The parallel superposition of \( E_1 \) and \( E_2 \) may be less costly than the evaluation of \( E_1 \) followed by the evaluation of \( E_2 \). Thus the parallel superposition is not only useful to express divide-and-conquer algorithms as shown in the next section, but it can also be used to efficiently program a parallel machine even without divide-and-conquer.

Using this operation, we can write a “scheduler” which composes several BSP programs (given for example by different users) in order to balance the sizes of the messages and decrease the number of synchronization barriers. If those programs are dynamically submitted to a queue, a formula describing their costs could be used by the “scheduler” to decide either to superpose several programs or to evaluate them sequentially. Superposing many programs would lead to too high values of \( h \). Moreover it is not a good idea to compose a program \( E_1 \) with a little amount of local computations and a high priority with a program \( E_2 \) with a lot of local computations because even if the superposition will be more efficient, the user who submitted the job \( E_1 \) will wait longer for its result. The same case occur if \( E_1 \) has few super-steps and \( E_2 \) many.

### 4.4 Example

The example presented below is a divide-and-conquer version of the scan program. The network is divided into two parts and the scan is recursively applied to those two parts. The value held by the last processor of the first part is broadcast to all the processors of the second part, then this value and the value held locally are combined together by the operator \( \text{op} \) on each processor of the second part.

```ocaml
let scan op vec =
let rec scan' fst lst op vec =
  if fst >= lst then vec else
  let mid = (fst + lst) / 2 in
  let vec' = mix mid (super (fun() -> scan' fst mid op vec)) (fun() -> scan'(mid + 1) lst op vec)) in
  let msg vec =
    apply (mkpar(fun i v-> if i = mid
```
then fun dst->if inbounds (mid+1) lst dst then Some v else None
else fun dst-> None) vec

and parop = parfun2(fun x y->match x with None->y|Some v->op v y) in
parop (apply(put(msg vec'))(mkpar(fun i->mid))) vec' in
scan' 0 (bsp_p()(-1) op vec

In this small program, we also use some functions which are parts of the current BSMLlib standard library:

let replicate f = mkpar(fun pid->f)

let parfun f v = apply (replicate f) v

as well as the following functions which will be in the standard library of BSMLlib with parallel superposition:

let mix m (v1,v2) =
  let f pid v1 v2 = if pid<=m then v1 else v2 in
  apply (apply (mkpar f) v1) v2

let inbounds first last n = (n->first)@@(n<=last)

This program was run on a sequential simulator. Nevertheless the BS\lambda-calculus with parallel superposition is confluent. Thus sequential and parallel evaluation give the same results.

5 Related Works

The libraries close to our framework based either on the functional language Haskell [26] or on the object-oriented language Python [19] propose flat operations similar to ours but no parallel composition.

[34] presents a way to divide-and-conquer in the framework of an object-oriented language. There is no formal semantics and no implementation from now on. The proposed operation is close to our parallel superposition (several BSP threads use the whole network) but the programmer has less control over the use of those super-threads. The same author advocates in [24] a new extension of the BSP model in order to ease the programming of divide-and-conquer BSP algorithms. It adds another level to the BSP model with new parameters to describe the parallel machine.

[36] is an algorithmic skeletons language based on the BSP model and offers divide-and-conquer skeletons. Nevertheless, the cost model is not really the BSP model but the D-BSP model [12] which allows subset synchronization. We follow [16] to reject such a possibility. Another algorithmic skeletons language based on the BSP model [17] does not offer divide-and-conquer skeletons. [31] presents another model which allows subset synchronization.

In the BSPlib library [18] subset synchronization is not allowed as explained in [32]. The PUB library [7] is another implementation of the BSPlib standard proposal. It offers additional features with respect to the standard which follows the BSP* model [4] and the D-BSP model [12]. Minimum spanning trees nested BSP algorithms [8] have been implemented using these features.

We also previously worked on a parallel composition [21]. This operation cannot be added to the BS\lambda-calculus (the obtained system is no longer confluent) because it is strategy dependent. To preserve a compositional cost model it must be used with an additional sync operation whose cost is \ell. Parallel superposition is thus the only one to propose a parallel composition which follows the simplest BSP model, which is compositional and which can be added to the BS\lambda_\ell-calculus.
6 Conclusions and Future Work

A parallel superposition has been added to the BSA/BSML11b framework. This new construction allows divide-and-conquer algorithms to be expressed easily, without breaking the BSP execution model.

Compared to a previous attempt [21], this new construction has not the drawbacks of its predecessor: the two sides of spatial parallel composition may not have the same number of synchronization barriers; the cost model is a compositional one; its semantics is not strategy dependent, it can be added to the BSA-calculus.

The next released implementation of the BSML11b library will include parallel superposition. Its ease of use will be experimented by implementing BSP algorithms described as divide-and-conquer algorithms in the literature.

Acknowledgments

This work is supported by the ACI Grid program from the French Ministry of Research, under the project CARAMI (http://www.carami.org).

References


