The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, January 2009, 1–20

RESEARCH ARTICLE

CPS Implementation of a BSP Composition Primitive with Application to the Implementation of Algorithmic Skeletons

Ilias Garnier[a] and Frédéric Gava[b*]

[a]*CEA, LIST, Mail Box 94, F91191 Gif-Sur-Yvette Cedex*;
[b]*LACL laboratory, University of Paris-East, France*

(*July 2009*)

BSML is an ML based language designed to code Bulk Synchronous Parallel (BSP) algorithms. It allows an estimation of execution time, avoids deadlocks and non-determinism. BSML proposes an extension of ML programming with a small set of primitives. One of these primitives, called parallel superposition, allows the parallel composition of two BSP programs. Nevertheless, its past implementation uses system threads and has a serious drawback which is the cost of managing threads in ML-like languages. This paper presents a new implementation of this primitive based on a continuation-passing-style (CPS) transformation guided by a flow analysis. To test it and show its usefulness, we also have implemented the OCamlP3L's algorithmic skeletons and compared their efficiencies with the original ones. The work presented here is tightly related to the BSP model, but is not specific to ML. Hence, we reckon there would be little work involved in translating it to, for instance, Java or Python.

Keywords: ML programming, BSP model, CPS transformation, flow analysis, skeletons

1. Introduction

1.1 Generalities

Since the paper "Go To Statement Considered Harmful", structured sequential programming is the norm. It is surprising to see that it is absolutely not the case for parallel programming [26]. Besides compiler-driven automatic parallelisation, programmers have kept the habit of using low-level parallel routines (as send/receive of *e.g.* MPI/PVM) or concurrent languages [32]. In this way, they, less or more, manage the communications with the usual problems of (un)buffered or (un)blocking sending, which are source of deadlocks and non-determinism. Furthermore, programmers forbid optimisations that could be done if more high-level structures as collective operators or skeletons were to be used instead.

High-level models, languages and tools are thus needed but sadly rarely used. The main reason of this fact is that they do generally not offer a sufficiently wide set of parallel structures (as skeletons [11]) for a practical and efficient programming.

This makes the design of new and robust parallel programming languages an important area of research. The design of these languages is a tradeoff between the possibility for the programmer to control parallel aspects necessary for predictable efficiency (but which make programs more difficult to write, to prove and

*Corresponding author. Email: frederic.gava@univ-paris-est.fr

ISSN: 1744-5760 print/ISSN 1744-5779 online

DOI: 10.1080/1744576YYxxxxxxxx
http://www.informaworld.com

to port) and the abstraction of such features which are necessary to make parallel programming easier — but which hampers efficiency and performance prediction.

1.2 Introduction to BSML

An interesting compromise is BSML, an extension of ML (a family of high-level programming languages: functional, modular and imperative) to code bulk-synchronous (BSP) algorithms [6] which combines the high degree of abstraction of ML (without poor performances because very often, ML programs are as efficient as C ones) with the scalable and predictable performances of BSP.

It aims at providing the right balance between the two opposite approaches to parallel programming: low-level programming which is subject to concurrency issues, versus high-level one with its loss of flexibility and efficiency. In the former, we find libraries such as MPI [40] generally used with Fortran or C; these approaches are unsafe and leave the programmer responsible for deadlock or non-determinism issues[1]. In the latter stand traditional algorithmic skeletons [11] where programs are safe but limited to a restricted set of algorithms.

BSP is a parallel model which offers an high degree of abstraction and allows an estimation of the execution time of its algorithms on a wide variety of architectures as massively parallel machines (Cray, *etc.*), beowulf clusters of multi/single-core PCs. This is particularly helpful in the design of efficient algorithms [4, 14] and routines in APIs [25]. In fact, many programs fit the BSP model even if many authors do not known it. For example, all the MPI programs that only use collective routines can be considered as BSP ones.

BSML is an extension of ML to code this kind of algorithms using a small set of primitives which are currently implemented as a parallel library (`http://bsmllib.free.fr`) for the ML programming language Objective Caml — OCaml. Using a safe high-level language as ML to program BSP algorithms allows performance, scalability and expressivity. BSML follows this paradigm to structure the computation and communication between the processors in a data-parallel fashion. All communications in BSML are collective (require all processes) and deadlocks are avoided by a strict distinction between local and global computation.

BSP also provides a simple and efficient cost model which is particularly helpful in the design of efficient algorithms [4, 6, 14] and that can be applied to BSML.

1.3 Superposition in BSML

The superposition primitive [33] is dedicated to the parallel composition of expressions (notably for divide-and-conquer algorithms) without any need of subset synchronisation[2]. It is based on sequentially interleaved threads of BSP computations, called *super-threads* [43]. Informally, it is equivalent to pairing in BSML.

It was shown in [20] that the parallel superposition is not only useful to divide-and-conquer BSP algorithms. This primitive can be used many times simultaneously in a single program and an efficient implementation is thus needed. To ensure a deterministic execution[3] of BSML programs, the semantics of the superposition forces to have, at any time, only one active super-thread.

---

[1]These properties are justified for concurrent computations but clearly not for parallel algorithms.
[2]Subset synchronisation of processors is usually justified by the necessity of the recursive decomposition of the computation into independent sub-problems; [43] argues that it is not really useful for BSP computing.
[3]Determinism guarantees that program behaviour is identical on all nodes; this essentially eliminates an entire class of errors: data races.

Currently, the super-threads are implemented over the system threads [19]. Scheduling system threads is costly [32], especially for a garbage-collected (GC) language as OCaml and when there are a lot of them. But we need to be able to create thousands of super-threads without incuring the performance hit that system threads cause. Otherwise, the superposition would lose its appeal. To overcome this problem, we present a novel implementation of the superposition which uses a global continuation-passing-style (CPS) transformation of BSML programs.

CPS is a classic style of programming in which control is passed explicitly in the form of a continuation [2]. Instead of "returning" values, a function takes an extra argument, the continuation which represents what should be done with the result of the function and then passes it to another function. Programs can be translated to semantically equivalent programs in CPS using a variety of algorithms [13]. As a programming device, CPS enables programmers to define advanced, application-specific control structures [42] such as co-routines [37, 45].

The main goal of this paper is a novel implementation of the superposition using a non-classical CPS transformation of BSML programs. We followed a pragmatic approach in the design of our global CPS transformation and efficiency was one of our major concerns[1]. Currently it works on a large subset of OCaml. This transformation is also driven by the use of a data flow analysis to introduce as less as possible the performance overhead of CPS — which is still less expensive than scheduling threads. Indeed, the superposition is transformed into CPS, whereas most of the code does not have to be modified.

To benchmark our transformation, we have tested it to the implementation of algorithmic skeletons, more precisely those of OCamlP3L [10] — `http://camlp3l.inria.fr/`. Skeletons languages are generally defined by introducing a limited set of parallel patterns to be composed in order to build easily a full parallel application [11]. Even if the implementation is less efficient compared to a dedicated skeleton language (or a MPI send/receive implementation [15]), the programmer can compose skeletons when it is natural for him and use a BSP programming style when it is necessary. Furthermore, as a performance test of our transformation, it has the advantage of generating a large number of super-threads.

1.4 Outline

A good familiarity with ML programming and type systems is assumed. We refer to the manual of OCaml (`http://caml.inria.fr/`, which also provides books for beginners) for a tutorial introduction to the language and links to type system papers. The approach we define here is not specific to ML though, and it could be applied to many strict high-level languages. We think there would be little work involved in adapting our system to Java or Python implementations of BSP [7, 27, 30].

First, we briefly review in Section 2 the BSP model, the BSML language and the past implementation of the superposition with its restrictions. We then give in Section 3 the definition (and some semantics results) of our novel CPS transformation which is used to implement efficient and scalable super-threads which are needed by the superposition. Implementation is described in Section 4. Section 5 is devoted to the benchmark of an implementation of OCamlP3L's skeletons using this transformation. Related work is discussed in Section 6.

[1] We know that OCaml code cannot be interrupted, so by adding an appropriate CPS in OCaml, we do not have to introduce some (inefficient) mechanism to save the execution context.

2. BSP Programming in ML

2.1 The Bulk-Synchronous Parallel Model

In the BSP model, a computer is a set of uniform processor-memory pairs and a communication network allowing the inter-processor delivery of messages [6, 39].

A BSP program is executed as a sequence of *super-steps* (see left scheme in Fig. 1), each one divided into three successive disjoint phases: each processor only uses its local data to perform sequential computations and to request data transfers to other nodes; the network delivers the requested data; a global synchronisation barrier occurs, making the transferred data available for the next super-step. The execution time (cost) of a super-step is the sum of the maximum of the local processing, the data delivery and the global synchronisation times.

The performance of the BSP machine is characterised by 4 parameters that can be benchmarked [6] to determine the execution time of BSP programs: the local processing speed **r**; the number of processors **p**; the time **L** required for a barrier; and the time **g** for collectively delivering a 1-relation, a communication phase where every processor receives/sends at most one word.

The network can deliver an h-relation (every processor receives/sends at most h words) in time $\mathbf{g} \times h$. The execution time (cost) of a super-step s is the sum of the maximal of the local processing, the data delivery and the global synchronisation times. The cost of a program is the total sum of the cost of its super-steps.

On most of today's distributed architectures, barrier synchronisations are often expensive when the number of processors dramatically increases. However, future shared memory architecture developments (such as multi-cores and GPUs) may make them much cheaper. They have also a number of attractions: it is harder to introduce the possibility of deadlock or livelock, since barriers do not create circular data dependencies. Barriers also permit novel forms of fault tolerance [39].

The BSP model considers communication actions *en masse*. This is less flexible than asynchronous messages, but easier to debug since there are many simultaneous communication actions in a parallel program, and their interactions are typically complex. Bulk sending also provides better performances since it is faster to send a block of data than individual ones — less network latency.

2.2 Bulk-Synchronous Parallel ML (BSML)

2.2.1 General description

BSML is currently a library based on the Objective Caml (OCaml) language; this choice was made among the different variants of ML available mainly for a reason of efficiency, since we target high-performance computation. Other reasons include the amount of libraries available and the tools provided. We plan a full language implementation by generating adequate OCaml code.

The core syntax of BSML is that of OCaml — with few restrictions. BSML programs can mostly be read as OCaml ones, in particular, the execution order should not seem unexpected to a programmer used to OCaml, even though the program is parallel. Moreover, most normal OCaml programs can be considered as BSML programs that do not make use of parallelism: the programs are executed sequentially on each processor of the parallel machine and return their results normally. That allows the parallelisation to be done incrementally from a sequential program.

Few entry points are needed for parallelism. BSML is based on a datatype called parallel vector which, among all OCaml types, enables parallelism. A vector has type `'a par` and embeds **p** values of type `'a` at each of the **p** different processors of the

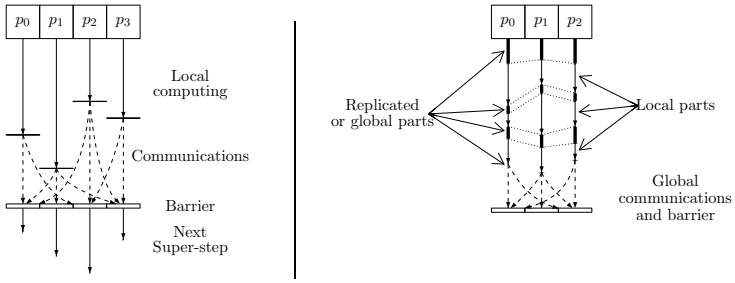


Figure 1. A BSP super-step (left) and BSML model of execution (right)

BSP machine. **p** is defined as a constant **bsp_p** throughout the execution of the program. We use the following notation to describe a parallel vector:$\langle x_0, x_1, \ldots, x_{\mathbf{p}-1} \rangle$.

2.2.2 Model of Execution

What distinguishes this structure from an usual vector of size **p** is that the different values, that will be called *local*, are blind from each other, as it is only possible to access the local value x_i in two cases: first locally, on processor i (by the use of a specific primitive) and second, after some communications.

These restrictions are inherent to distributed memory parallelism; here they are enforced by the use of an opaque type. This choice also makes parallelism fully explicit, BSP costs easier to analyse [21] and we think programs more readable. Worth noting is that parallel vectors can not be embedded in themselves since the BSP machine has only one level of parallelism. We refer to [22, 23] for discussions on the problematic BSP implementation of nested parallel vectors.

Since a BSML program deals with a whole parallel machine and individual processors at the same time, a distinction between the levels of execution that take place will be needed (see right scheme in Fig. 1):

- **Replicated** execution is the default. Code that does not involve BSML primitives (nor, as a consequence, parallel vectors) is run by the parallel machine as it would be by a single processor. It is used to coordinate the work of each processors.
- **Local** execution is what happens inside parallel vectors, on each of their components: the processor uses its local data to do computations that may be different from the others. Replicated and Local execution are strictly disjoint, and typically, processors alternate between them.

The distinction between local and replicated is strict [22, 23]. Hence, the replicated code can not depend on local information, and remains replicated.

2.2.3 Parallel primitives

Parallel vectors are handled through the use of different communications primitives that constitute the core of BSML. Their implementation relies either on MPI, PUB [8] or on the TCP/IP functions provided by OCaml. A toplevel is also provided where the user can define its number of processors: execution on the toplevel or on a parallel machine gives the same results — except in time.

Fig. 2 subsumes the use of the primitives. Informally, primitives works as follows. Let $\ll \mathsf{x} \gg$ be the vector holding x everywhere — on each processor. The $\ll \gg$ indicate that we enter a local section and pass to the local level. Replicated information is available inside the vector. Now, to access local information, we add the syntax x to open the vector x and get the local value it contains, which can obviously be used only within local sections.

The **proj** primitive is the only way to extract a local value from a vector. Given a vector, it returns a function such that applied to the pid of a processor, it returns

primitive	type	informal description
≪ e ≫	t par (if $e : t$)	$\langle e, \ldots, e \rangle$
pid (within a vector	int	i on processor i
v (within a vector)	t (if $v : t$ par)	v_i on processor i (if $v = \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$)
proj	'apar→ (int→ 'a)	$\langle x_0, \ldots, x_{p-1} \rangle \mapsto (\mathsf{fun}\, i \to x_i)$
put	(int→ 'a)par→ (int→ 'a)par	$\langle f_0, \ldots, f_{\mathbf{p}-1} \rangle \mapsto \langle (\mathsf{fun}\, i \to f_i\ 0), \ldots, (\mathsf{fun}\, i \to f_i\ (\mathbf{p}-1)) \rangle$
super	(unit→ 'a)→ (unit→ 'a)→ 'a∗'b	$f_a \mapsto f_b \mapsto (f_a\ (), f_b\ ())$

Figure 2. Summary of BSML primitives

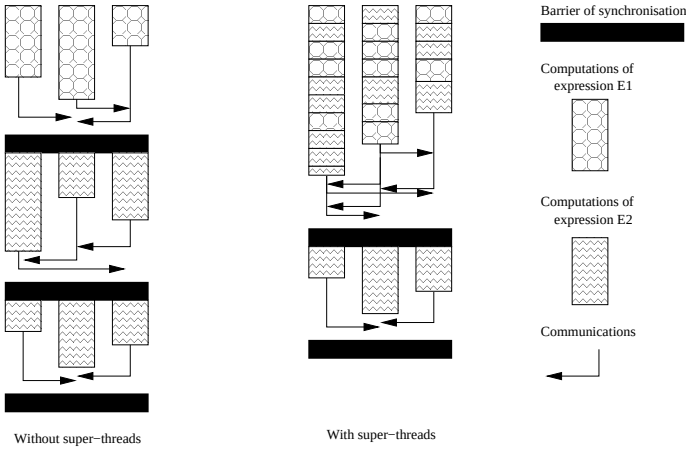


Figure 3. The evaluation of the superposition for two super-threads on 3 processors

the value of the vector at this processor. **proj** performs communications to make local results available globally within the returned function. Hence it establishes a meeting point for all processors and, in BSP terms, ends the current super-step.

Note the choice of functions of type (int → 'a) in **proj**. Arrays of size **p** or lists could have been chosen instead, but the interface is more functional and generic this way. Furthermore, as seen in the examples, the conversion between one style and the other is easy. Internally, our implementation relies on arrays.

The **put** primitive is the comprehensive communication primitive. It allows any local value to be transferred to any other processor. As such, it is more flexible than **proj**. It is as well synchronous, and ends the current super-step.

The parameter of **put** is a vector that, at each processor, holds a function of type (int → 'a) returning the data to be sent to processor i when applied to i. The canonical use of **put** is: (**put** ≪ **fun** sendto → e(pid, sendto, x) ≫) where expression e computes (or usually, selects) the data that should be sent depending on sender to sendto. The result of **put** is another vector of functions that return, when applied to i, the value *received from* processor i.

BSP paradigm's simplicity and elegance comes at a cost: the ability to synchronise a subset of the processors would break the BSP cost model. Subset synchronisation is generally used to recursively decompose computations into independent tasks — the divide-and-conquer paradigm. However, [43] proposes a natural way to fit divide-and-conquer algorithms into the BSP framework without subset synchronisation and by using sequentially interleaved threads of BSP computation, called *super-threads*. The last primitive called **super** allows the evaluation of two BSML expressions E_1 and E_2 as super-threads [17, 34]. From the programmer's point of view, the semantics of the superposition is the same as pairing *i.e.*, building the pair (E_1, E_2) but of course the evaluation of **super** $E_1\ E_2$ is different — see Fig. 3.

In the left, pairing is just the sequential evaluation of the two programs. In the right, using the superposition, the phases of asynchronous computation of E_1

and E_2 are run. Then the communication phase of E_1 is merged with that of E_2. The messages are obtained by simple concatenation of the messages of each super-thread and only one barrier occurs. If the evaluation of E_1 needs more super-steps than that of E_2 then the evaluation of E_1 continues — and *vice versa*. The parallel superposition is thus less costly than the evaluation of E_1 followed by E_2.

2.2.4 Other features

BSML can also be used to manage external memories (file systems) [18]. This feature is necessary to have safe accesses to local files (on each processor) and global ones — available to all processors as in MPI-I/O. Local files can not be accessed in replicated environment and *vice-versa* for global ones. [24] presents how to manage exceptions in BSML. Currently, we do not have extended this work to handle the contents of this article. We will discuss this point in the conclusion.

2.3 Examples

Having a very small core of parallel operations is a great strength for the formalisation of the language. It makes the definitions clear and the proofs shorter. However, the use of the primitives can sometimes become awkward. Defining some useful libraries simplify the coding of the algorithms. In this section, we define a few functions useful for BSP computing where some are given as additional BSML libraries. They are typical examples of BSML programming.

2.3.1 Utility functions

It is often needed to split a dataset among the processors. The following function is an example that selects a part of a list on every processor:

```
(* select_list:'a list → 'a list par *)
let select_list l = let len = List.length l
                    in ≪ cut_list l ($pid$ * len / bsp_p) (($pid$ + 1) * len / bsp_p))≫
```

where cut_list l a b returns the sub-list of the elements of l from index a (inclusive) to index b — exclusive. Now, to parallel map a function on a list (classical data-parallel skeleton [1, 11]) scattered as above is as simple as:

```
(* parmap : ('a→ 'b)→ 'a list par → 'b list par *)
let parmap f parlist = ≪ (List.map f) $parlist$≫
```

The **proj** primitive is often used at the end of a parallel computation to gather the computed results. For example, converting a parallel vector into a list:

```
(* proj_list : 'a par → 'a list *)
let proj_list v = List.map (proj v) procs_list
```

where procs_list is the list of pids: $[0; 1; \ldots \mathbf{p}-1]$.

2.3.2 Parallel prefix computation

As a generalisation of the above, a simple one-step reduce (having $\oplus_{k=0}^{\mathbf{p}-1} v_k$ on each processor from the parallel vector $\langle v_0, v_1, \ldots, v_{\mathbf{p}-1} \rangle$) can be done with:

```
(* simple_reduce: ('a→ 'a→ 'a)→ 'a→ 'a par → 'a
let simple_reduce op e v = List.fold_left op e (proj_list v)
```

where e is a neutral element and op the associative operator $\oplus$.

The above reduce does not make use of parallelism. If the combination operator $\oplus$ has some cost, we may prefer to reduce in a multi-step manner (a classic logarithmic way), doing the combinations locally. It is based on the classic parallel prefix computation: scan $e \oplus$ $\boxed{v_0}\boxed{\cdots}\boxed{v_{\mathbf{p}-1}}$ $=$ $\boxed{e}\boxed{v_0 \oplus v_1}\boxed{v_0 \oplus v_1 \oplus v_2}\boxed{\cdots}\boxed{\oplus_{k=0}^{\mathbf{p}-1} v_k}$

This algorithm combines the values of processors i and $i + 2^n$ at processor i for every super-step n from 0 to $\lceil \log_2(\mathbf{p}) \rceil$. Fig. 4 (left) gives the code.

```
(* scan': int→ ('a→ 'a→ 'a)→ 'a→ 'apar→ 'apar *)
let rec scan' step op e v =
 if step >= bsp_p then v else
  let comm =
     put ≪ fun j →
               if (j mod (2*step) = 0)
                 && ($pid$ = j + step)
                then $v$
               else e ≫
   in let v' =
        ≪ if $pid$ mod (2*step) = 0
             then
              if $pid$ + step < bsp_p
               then op $v$ ($comm$ ($pid$ + step))
              else $v$
          else e ≫
 in scan' (step*2) op e v'

(* reduce: ('a→ 'a→ 'a)→ 'a→ 'a par→ 'a *)
let reduce op e v = (proj (scan' 1 op e v)) (bsp_p−1)
```

```
(* inbounds: 'a→ 'a→ 'a→ bool *)
let inbounds first last n = (n>=first)&& (n<=last)
(* mix: int→ 'a par * 'a par→ 'a par *)
let mix m (v1,v2) = ≪ if $pid$<=m then $v1$ else $v2$ ≫

(* scan_super: ('a→ 'a→ 'a)→ 'a→ 'a par→ 'a par *)
let scan_super op e vec =
 let rec scan' fst lst vec =
  if fst>=lst then vec
  else
   let mid = (fst+lst)/2 in
   let vec'= mix mid (super (fun()→ scan' fst mid vec)
                            (fun()→ scan'(mid+1) lst vec)) in
   let send = put ≪ if $pid$=mid
                    then (fun dst→ if inbounds (mid+1) lst dst
                                   then Some $vec'$
                                  else None)
                    else (fun dst→  None)≫ in
     ≪ match ($send$ mid) with
         | None → $vec'$
         | Some v → op v $vec'$ ≫
 in scan' 0 (bsp_p()−1) vec)
```

Figure 4. Parallel reduce computation (left) and its Divide-and-conquer version (right)

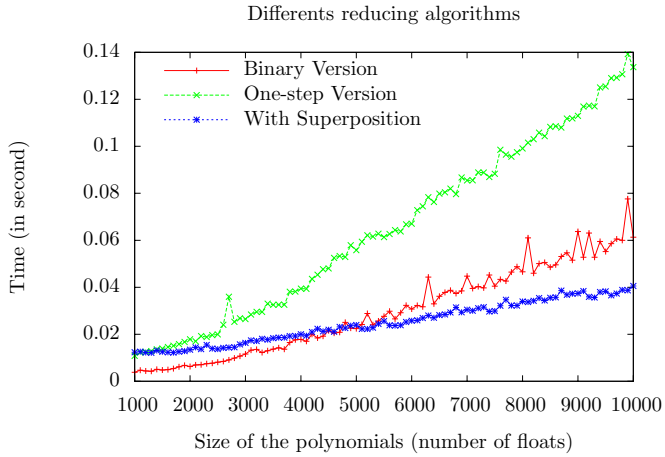


Figure 5. Benchmarking of reducing algorithms (addition of polynomials)

The program (scan' 1) gathers data at every even processor, then at multiples of 4, 8, *etc.* Communication is the first step: the argument of put returns the operator's unit unless sending from processor $(2 \times step + 1) \times i$ to $2 \times step \times i$ for any i. Then, the combination is done at processor $2 \times step \times i$ using op.

In the same manner, the parallel prefix computation can be done using a divide-and-conquer BSP algorithm (with **super**) where the processors are divided into two parts and the scan is recursively applied to those 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 values held locally are combined together by the associative operator $\oplus$ on the second part. Fig. 4 (right) gives the code of this method.

Fig. 5 presents some experiments on a cluster with 10 Pentium IV nodes (1 GB of RAM each) interconnected with a Gigabyte Ethernet network and Open-MPI. The binary version is the one presented in Fig. 4 (left) and superposition is the one in the right of Fig. 4. The values were arrays of floats representing polynomials and the binary operation is the sum. Diagrams show the average of the results with increasing size of polynomials. Note that the performances for the superposition version were done using both past and new implementations and no difference was found here due to a too small number of super-threads.

2.4 Older superposition implementation

In ML-like languages, it is straightforward to add imperative features that can introduce non-deterministic results (deadlocks) in BSML. To avoid this, a strategy for the choice of the unique active super-thread has been added [19]: the active super-thread is evaluated until it ends its computations or it needs communications. When communications are done, the first super-thread which has finished "its past super-step" is re-evaluated, *i.e.*, it becomes the new current active super-thread.

Currently, based on a semantics study, the superposition is implemented using system threads [19]. Each time a superposition is called, a new thread is created and share locks are used each time a communication primitive is called.

There is an important drawback to this method. System threads slow down the running of an OCaml program: a global lock is used due to the GC of OCaml. When using many threads (*e.g.* > 100), the performance of the program totally collapses. This greatly limits the usefulness of the superposition when a large number of super-threads are run simultaneously. We now present another implementation which uses a global continuation-passing-style (CPS) transformation.

3. CPS transformation and flow analysis

First, we present a naive CPS transformation and the core source language which is CoreML with the adjunction of two primitives: **yield** and **super** — the target language is the same minus these primitives. Here, **yield** replaces communication primitives, abstracting away communication handling. **yield** suspends the currently executing super-thread (called thread in the next) and schedules the execution of the next thread, as defined by the semantics of **super**. We then define our novel CPS transformation which aims to preserve as much code as possible.

3.1 Continuation Passing Style

The original CPS transform [35] was designed to study the various evaluation strategies for the lambda-calculus by making the control explicit, as a *continuation*: a function representing the evaluation context. It was then discovered that giving to the programmer or the compiler writer the ability to explicitly manipulate continuations was an expressive tool to perform various analysis or to encode various high-level constructs, such as exceptions or light-weight threads [37]. Below is the original CPS transformation, as defined in [35] for λ-calculus:

$$\begin{array}{ll} [\![x]\!] & = \lambda k.(k\ x) \\ [\![\lambda x.M]\!] & = \lambda k.(k\ (\lambda x.[\![M]\!])) \\ [\![(M\ N)]\!] & = \lambda k.([\![M]\!]\ (\lambda m.([\![N]\!]\ (\lambda n.(m\ (n\ k)))))) \end{array}$$

3.2 Core language and naive CPS transformation

Expressions are as follow (values v are defined as the usual subset):

$e ::=$	x	*variables*	|	$\kappa\ e$	*constructor application*
|	c	*constants*	|	**op** $e_1\ e_2$	*arithmetic operators*
|	$\lambda v.e$	*functional values*	|	**super** $e_1\ e_2$	*superposition*
|	**fix** $f\ \lambda x.e$	*recursive functions*	|	$\langle e \rangle$	*parallel vector*
|	$e_1\ e_2$	*applications*	|	$\$x\$$	*get the local value of a vector*
|	**let** $v = e_1$ **in** e_2	*local definitions*	|	**yield**	*simulates* **put** and **proj**
|	(e_1, e_2)	*couples*			
|	**match** e **with** $m_1 \mid ... \mid m_n$	*pattern matching*			
$m ::=$	$\kappa\ x \rightarrow e$	*matching branch*			

Monads allow to extend a language while enforcing a correct operational behaviour [44]. A monad is the data of three primitives: **run**, **ret** and **bind**, operating

$$
\begin{array}{l|l}
T_0[\![x]\!] = \mathbf{ret}\ x & T_0[\![\mathbf{fix}\ f\,\lambda x.e]\!] = \mathbf{ret}\ (\mathbf{fix}\ f\,\lambda x.T_0[\![e]\!]) \\
T_0[\![\$x\$]\!] = \mathbf{ret}\ \$x\$ & T_0[\![e_1\ e_2]\!] = \mathbf{bind}\ T_0[\![e_1]\!]\ (\lambda v_1.\mathbf{bind}\ T_0[\![e_2]\!]\ (\lambda v_2.v_1\ v_2)) \\
T_0[\![c]\!] = \mathbf{ret}\ c & T_0[\![\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2]\!] = \mathbf{bind}\ T_0[\![e_1]\!]\ (\lambda v.T_0[\![e_2]\!]) \\
T_0[\![\lambda v.e]\!] = \mathbf{ret}\ \lambda v.T_0[\![e]\!] & T_0[\![(e_1,e_2)]\!] = \mathbf{bind}\ T_0[\![e_1]\!]\ (\lambda v_1.\mathbf{bind}\ T_0[\![e_2]\!]\ (\lambda v_2.(v_1,v_2))) \\
T_0[\![\langle e\rangle]\!] = \mathbf{ret}\ \langle T_0[\![e]\!]\rangle & T_0[\![\kappa\ e]\!] = \mathbf{bind}\ T_0[\![e]\!]\ (\lambda v_e.\kappa\ v_e) \\
 & T_0[\![\mathbf{op}\ e_1\ e_2]\!] = \mathbf{bind}\ T_0[\![e_1]\!]\ (\lambda v_1.\mathbf{bind}\ T_0[\![e_2]\!]\ (\lambda v_2.op\ v_1\ v_2))
\end{array}
$$

$$
T_0[\![\mathbf{match}\ e\ \mathbf{with}\ |\ \kappa_i\ x_i \rightarrow\ e_i\ ]\!] = \mathbf{bind}\ T_0[\![e]\!]\ (\lambda v_e.\mathbf{match}\ v_e\ \mathbf{with}\ |\ \kappa_i\ x_i \rightarrow\ T_0[\![e_i]\!])
$$

$$
\begin{array}{l}
\mathbf{yield} = \lambda k.\mathbf{Waiting}\ k \\
\mathbf{super} = \mathbf{let}\ loop = \mathbf{fix}\ loop\,\lambda r_1.\lambda r_2.\mathbf{bind}\ \mathbf{yield}\ (\lambda\,().\mathbf{match}\,(r_1,\ r_2)\ \mathbf{with} \\
\qquad |\ (\mathbf{Terminated}\,x_1,\ \mathbf{Terminated}\,x_2) \rightarrow \mathbf{ret}\ (x1,\ x2) \\
\qquad |\ (\mathbf{Terminated}\,\_,\ \mathbf{Waiting}\,s) \rightarrow loop\ r_1\ (s\ ()) \\
\qquad |\ (\mathbf{Waiting}\,s,\ \mathbf{Terminated}\,\_) \rightarrow loop\ (s\ ())\ r_2 \\
\qquad |\ (\mathbf{Waiting}\,s_1,\ \mathbf{Waiting}\,s_2) \rightarrow loop\ (s_1\ ())\ (s_2\ ()))\ \mathbf{in} \\
\qquad\qquad \mathbf{ret}\ \lambda f_1.\mathbf{ret}\ \lambda f_2. \\
\qquad\qquad\quad \mathbf{let}\,r_1 = ((\mathbf{ret}\ f_1)\,@\,(\mathbf{ret}\ ()))\ (\lambda x.\mathbf{Terminated}\,x)\ \mathbf{in} \\
\qquad\qquad\quad \mathbf{let}\,r_2 = ((\mathbf{ret}\ f_2)\,@\,(\mathbf{ret}\ ()))\ (\lambda x.\mathbf{Terminated}\,x)\ \mathbf{in} \\
\qquad\qquad\quad\ loop\ r_1\ r_2
\end{array}
$$

Figure 6. Monadic transformation

on a type $M\,\alpha$. **run** has type $\forall\alpha.M\,\alpha{\rightarrow}\alpha$ and executes a monadic program. **ret**, of type $\forall\alpha.\alpha{\rightarrow}M\,\alpha$ transforms a base value into a monadic one. Finally, **bind** allows chaining monadic computation as reflected by its type $\forall\alpha,\beta.M\,\alpha{\rightarrow}(\alpha{\rightarrow}M\,\beta){\rightarrow}M\,\beta$.

Our threads are modelled as resumptions, meaning that they are in a suspended state or terminated: `type 'a thread=Terminated of 'a | Waiting of (unit→ 'a thread)` The monadic type is always *thread*: $M\,\alpha = \forall\,\beta.(\alpha \rightarrow thread\,\beta) \rightarrow thread\,\beta$. The monadic primitives are thus defined as follow:

$$
\begin{array}{lcl}
\mathbf{ret}\ \mathrm{x} & = & \lambda k.kx \\
\mathbf{bind}\ \mathrm{m\ f} & = & \lambda k.\mathrm{m}(\lambda v.fvk) \\
\mathbf{run} & = & \lambda x.((\mathbf{fix}\ loop\,\lambda t.\ \mathbf{match}\ t\ \mathbf{with} \\
 & & \qquad |\ \mathbf{Terminated}\ x \rightarrow\ x \\
 & & \qquad |\ \mathbf{Waiting}\ s \rightarrow\ loop\ (s\ ()))\ (x\ (\lambda x.\mathbf{Terminated}\ x)))
\end{array}
$$

They must at least satisfy these three monadic laws:

$$
\begin{array}{lcl}
\mathbf{bind}\ (\mathbf{ret}\ a)\ f & \approx & f\ a \\
\mathbf{bind}\ a\ \lambda x.\mathbf{ret}\ x & \approx & a \\
\mathbf{bind}\ (\mathbf{bind}\ a\ (\lambda x.b))\ (\lambda y.c) & \approx & \mathbf{bind}\ a\ (\lambda x.\mathbf{bind}\ b\ (\lambda y.c))
\end{array}
$$

where $\approx$ is defined as $\forall a_1, a_2, k\,\exists a\,(a_1\,k \Rightarrow a){\wedge}(a_2\,k \Rightarrow a)$ and $\Rightarrow$ is a simple big-step semantics [16]. In our case, these laws were mechanically proved using the Coq proof assistant — see [16] for the proof script and for all other proofs.

We now straightforwardly proceed to the definition of the monadic transformation on expressions $T_0[\![e]\!]$, defined in Fig. 6 — left part. The primitives **yield** and **super** are then defined using first class continuations in Fig. 6 (right part) where $a\,@\,b \equiv \mathbf{bind}\ \ a\ (\lambda v_a.\mathbf{bind}\ \ b\ (\lambda v_b.(v_a\ v_b)))$.

The operational behaviour of these primitives is clear: **yield** captures its own continuation, and stores it into a suspension for further evaluation; **super** first suspends its own execution (using **yield**), then schedules the execution of its two sub-threads until they are terminated. We have the following correction result:

THEOREM 3.1 *If* $e \Rightarrow v$ *then* $T_0[\![e]\!] \approx$ ***ret*** v.

3.3 Flow analysis for performance issues

To illustrate the problem of this naive transformation, we give this trivial example:

$$
\begin{array}{lcl}
\mathbf{apply} & = & \lambda f.\lambda x.fx \\
T_0[\![apply]\!] & = & \lambda k_0.k_0\lambda f.\lambda k_1.k_1\lambda x.(\lambda k_2.(\lambda k_3.k_3 f)(\lambda v.(\lambda v_f.\lambda k_4.(\lambda k_5.k_5 x)(\lambda v.(\lambda v_x.v_f v_x)vk_4))vk_2))
\end{array}
$$

It is obvious that this transformation can not be used as is. Moreover, this code did not need to be converted to CPS at all: it does not contain any concurrency primitive. The full transformation of a program to CPS considerably impedes performance. This overhead is usually alleviated using transformation-time

reductions to eliminate these so-called "administrative redexes" on the programs.

However, that does not suffice. Aiming at numerical computing, we can not afford to transform unnecessary expressions. Preserving these kind of expression from being converted is the aim of the transformation which is now presented. Observing how some very limited parts of the program need continuations, it seems natural to try to convert only the required expressions — in our case, only **yield** and **super** need them. We thus need a partial CPS transformation [31]. The expressions to be transformed are those susceptible to reduce a **yield** or **super** expression.

Since we must cope with higher-order functions, the partial CPS transformation is guided by a flow analysis which yields a straightforward flow inference algorithm whose purpose is to decide if an expression is susceptible to reduce a **yield**: we tag it as *impure* — *pure* otherwise. Our type system is derived from the type system for CFA defined in [28] and the rules can be found in Fig. 7 where τ are classic ML types, and flows F ::= $\mathcal{P}$ | $\mathcal{I}$ — pure or impure where $\mathcal{I} < \mathcal{P}$. We also suppose that the nesting of parallel vectors is impossible after the analysis described in [23]. We use ground, simple types τ annotated by flows F:

$$\begin{array}{lll} \tau ::= \langle F, CstT\rangle & & \\ \quad | \quad \langle F, TName\rangle \text{ user-defined sum types} & & | \ \langle F, \tau_0 * \tau_1\rangle \text{ couples} \\ \quad | \quad \langle F, \tau_0 \to \tau_1\rangle \text{ functions} & & | \ \langle F, \tau \ par\rangle \text{ type of vectors} \\ \multicolumn{3}{c}{CstT \ ::= \ unit \ | \ int \text{ constants}} \end{array}$$

Each constructor κ has a domain type (the type of its argument) and a codomain type — the *typename* to which κ belongs. These are denoted κ_{dom} and κ_{codom}. We also define two projection functions on types : $annot(\langle f, x\rangle) = x$ and $\text{flow}(\langle f, x\rangle) = f$. These functions are readily extended to typed source terms. For any expressions a and b, we define $a \vee b = min(flow(a), flow(b))$.

Before stating the soundness theorem, we will state the usual lemma on type-preserving (and thus *flow preserving*) substitutions:

LEMMA 3.2 (Typings are stable by substitution)
Let e be an expression such that $\Gamma, x : \tau \vdash e : \tau'$ holds, and let v be an expression such that $\Gamma \vdash v : \tau$. Then $\Gamma \vdash [v/x]e : \tau'$.

THEOREM 3.3 (Soundness w.r.t. **yield** reductions)
*If $\Gamma \vdash e : \tau$ and $e \Rightarrow v$ then $\Gamma \vdash v : \tau$. If e contains **yield** then $flow(e) = \mathcal{I}$.*

3.4 Partial CPS transformation

The flow-directed partial CPS transformation aims to preserve "pure" expressions, while CPS-converting "impure" ones. To preserve the operational equivalence, some code must be generated between CPS and non-CPS terms.

The partial transformation is simple, parallel primitives are directly replaced by their definitions, and the operators are always pure. Therefore pure expressions are preserved from being transformed. On real-world programs, most of the computation takes place in pure expressions, making the CPS part less of a burden:

$$\begin{array}{ll} T_1[\![x]\!] & = \textbf{ret } x \\ T_1[\![\$x\$]\!] & = \textbf{ret } \$x\$ \\ T_1[\![c]\!] & = \textbf{ret } c \\ T_1[\![\lambda v.e^{\mathcal{I}}]\!] & = \textbf{ret } \lambda v.T_1[\![e]\!] \\ T_1[\![(\textbf{fix } h\, \lambda x.e)^{\mathcal{I}}]\!] & = \textbf{ret } \textbf{fix } h\, \lambda x.T_1[\![e]\!] \end{array}$$

$$\begin{array}{ll} T_1[\![(e_1\ e_2^{\mathcal{P}})^{\mathcal{I}}]\!] & = T_1[\![e_1]\!](\textbf{ret } e_2) \\ T_1[\![(\textbf{let } v = e_1^{\mathcal{I}} \textbf{ in } e_2)^{\mathcal{I}}]\!] & = \textbf{bind } T_1[\![e_1]\!]\ (\lambda v.T_1[\![e_2]\!]) \\ T_1[\![(\kappa\ e)^{\mathcal{I}}]\!] & = \textbf{bind } T_1[\![e]\!]\ (\lambda v_e.\textbf{ret } \kappa\, v_e) \\ T_1[\![(\textbf{let } v = e_1^{\mathcal{P}} \textbf{ in } e_2)^{\mathcal{I}}]\!] & = \textbf{let } v = e_1 \textbf{ in } T_1[\![e_2]\!] \\ T_1[\![e^{\mathcal{P}}]\!] & = \textbf{ret } e \\ T_1[\![\langle e^{\mathcal{P}}\rangle]\!] & = \textbf{ret } \langle e\rangle \end{array}$$

$$\begin{array}{ll} T_1[\![(e_1\ e_2^{\mathcal{I}})^{\mathcal{I}}]\!] & = \textbf{bind } T_1[\![e_1]\!]\ (\lambda v_1.\textbf{bind } T_1[\![e_2]\!]\ (\lambda v_2.v_1 v_2)) \\ T_1[\![(e_1,\ e_2)^{\mathcal{I}}]\!] & = \textbf{bind } T_1[\![e_1]\!]\ (\lambda v_1.\textbf{bind } T_1[\![e_2]\!]\ (\lambda v_2.\textbf{ret } (v_1, v_2))) \\ T_1[\![\textbf{match } e \textbf{ with } |\ \kappa_i\ x_i \to\ e_i\ ]\!] & = \textbf{bind } T_1[\![e]\!]\ (\lambda v_e.\textbf{match } v_e \textbf{ with } |\ \kappa_i\ x_i \to\ T_1[\![e_i]\!]) \end{array}$$

We observe that an impure expression is never embedded into a pure one. This property is induced by the type system: if any sub-expression e_i of an expression

$$\frac{x:\tau\in\Gamma}{\Gamma\vdash x:\tau}\quad\frac{x:\tau\ par\in\Gamma}{\Gamma\vdash \$x\$:\tau}\quad\frac{}{\Gamma\vdash c:\langle\mathcal{P},CstT\rangle}\quad\frac{}{\Gamma\vdash op:\tau_{op}}\quad\frac{\Gamma\vdash e:\langle\mathcal{P},\tau\rangle}{\Gamma\vdash\langle e\rangle:\langle\mathcal{P},\tau\ par\rangle}\quad\frac{\Gamma,x:\tau_1\vdash z:\tau_2\ \text{ if } f{=}x\vee z}{\Gamma\vdash\lambda x.z:\langle f,\tau_1\to\tau_2\rangle}$$

$$\frac{\Gamma\vdash z\ \ \Gamma\vdash z':\tau_1\quad\text{if } annot(z){=}\tau_1\to\tau_2 \text{ and } f{=}z\vee z'}{\Gamma\vdash(z\,z'):\langle f,annot(\tau_2)\rangle}\quad\frac{\Gamma\vdash z:\tau_1\qquad\Gamma\vdash z':\tau_2\quad\text{if } if{=}z\vee z'}{\Gamma\vdash(z,z'):\langle f,\tau_1*\tau_2\rangle}$$

$$\frac{\Gamma\vdash z:\tau_1\ \ \Gamma,x:\tau_1\vdash z':\tau_2\quad\text{if } flow(\tau_2)\leq_F flow(\tau_1)}{\Gamma\vdash\mathbf{let}\,x\,=\,z\,\mathbf{in}\,z':\tau_2}\quad\frac{\Gamma,h:\langle f,\tau_0\to\tau_1\rangle,x:\tau_0\vdash z:\tau_1\quad\text{if } f{=}x\vee z}{\Gamma\vdash\mathbf{fix}\,h\,\lambda x.z:\langle f,\tau_0\to\tau_1\rangle}$$

$$\frac{\Gamma\vdash e:\tau_\kappa\ \ \kappa:\tau_\kappa\to\langle flow(\tau_\kappa),\mu\,\alpha\,.\,\oplus_{i=1}^n\tau_{\kappa_i}\rangle}{\Gamma\vdash\kappa\,e:\langle flow(\tau_\kappa),\mu\,\alpha\,.\,\oplus_{i=1}^n\tau_{\kappa_i}\rangle}\quad\frac{\Gamma\vdash e:\langle F,\mu\,\alpha\,.\,\oplus_{i=1}^n\tau_{\kappa_i}\rangle\ \ \Gamma,x_i:\tau_{\kappa_i}\vdash e_i:\tau\ \ i\in[1\cdots n]}{\Gamma\vdash\mathbf{match}\,e\,\mathbf{with}\ \mid\kappa_i\,x_i\ \to\ e_i:\tau}$$

$$\frac{\text{if } f{=}min(flow(\tau_0),flow(\tau_1))}{\Gamma\vdash\mathbf{super}:\langle\mathcal{I},\langle\mathcal{I},\langle\mathcal{P},unit\rangle\to\tau_0\rangle\to\langle\mathcal{I},\langle\mathcal{I},\langle\mathcal{P},unit\rangle\to\tau_1\rangle\to\langle f,\tau_0*\tau_1\rangle\rangle\rangle}\quad\frac{}{\Gamma\vdash\mathbf{yield}:\langle\mathcal{I},unit\rangle}$$

Figure 7. Inference rules

e is impure, so is e. We use this fact in the transformation: when encountering a pure expression, we simply wrap it into a **ret**. Our type system enforces that all variables bound to the same binder have the same flow, ensuring the soundness of our framework. This allows us to prove some useful lemmas on substitutions that make the following soundness theorem provable:

THEOREM 3.4 *If* $t \Rightarrow v$ *then* $T_1[\![t]\!] \approx \boldsymbol{ret}\ \ v$.

4. New implementation of the superposition

For lack of space, we do not give all the details. The full implementation (which works as a source-to-source code transformer) is available at `http://lacl.univ-paris12.fr/gava/cps-super-bsml-comp.tar.gz`. Currently our implementation works on a large subset of OCaml without objects, labels and functors.

4.1 Treatment of OCaml expressions

4.1.1 Imperative features, Sum type and records

We did not treat imperative features here, suffice to say that every expression involved in an imperative operation is constrained to have the flow of its sub-expressions. When encountering an impure loop, we must convert it into its tail-recursive equivalent form. OCaml handles tail-recursion fine, so there is no added risk of stack overflow. The soundness of this transformation was not proved, but we have not encountered any problems with them so far.

Our type-based flow analysis handles sum types and records. The possible issues arise when we want to transform them: our backend language, OCaml, is strongly typed, and our transformation totally changes the types of impure expressions. Two solutions were envisioned: effectively disabling the type-checker by using type casts on each data constructor or modifying the type declarations accordingly, which has as an added advantage the possibility of having better performances thanks to more precise type information. We chose the second solution.

4.1.2 Defunctorisation

OCaml provides parametric modularity (known as *functors*), but our transformation does not handle them. In order to apply our transformation, we have to defunctorise the whole program. To this end, we use the technique described in [38]. A nice side-effect is the increased possibilities in inlining and code specialisation.

4.2 Generation of the code

4.2.1 Monomorphisation

Our partial CPS transformation needs simple types. Thus, we need to monomorphise the whole program. After type inference, the syntax tree is annotated with either ground types or type schemes, which are introduced only at **let** bindings. Each of these bindings is possibly instantiated with different types. Monomorphisation is the act of duplicating these bindings for each instantiation type — duplicating polymorphically typed functions for each needed domain type. We must also specialise type declarations to take into account impure functions: we scan the whole program, registering each type used inside algebraic data constructors or records and instantiate declarations accordingly. We must then perform a topological sort to take into account the fact that a polymorphic type may be used with a type declared after. Monomorphisation can potentially make the size of the program grow exponentially, but actual implementations (as MLton, `http://mlton.org/`) shows that practically, the size growth is manageable — about 30 %.

4.2.2 Monoflowisation

In fine, to maximize the efficiency of the generated code, we use a similar process for flows: instead of duplicating functions based on types, we duplicate them based on flows. This is of utmost importance for widely used functions: if they are used with an impure argument throughout the code, they are flagged as impure for *every* call site even with pure arguments. This is a consequence of our flow analysis being monovariant. The monoflowisation is performed directly during the monomorphisation. Another solution would be to use a polyvariant flow analysis, but it would be extremely heavy, both in algorithmic complexity and in implementation.

4.2.3 Code duplication

Since we are typing the whole program, we know exactly each instantiation type for each binding, allowing us to create as many ground versions of them as we need. In order to avoid variable capture problem, we bind each specialised code to a fresh name, and update the instantiation points accordingly. The freshness is ensured by performing an alpha-conversion pass on the whole program before monomorph(*flow*)isation. Monomorphisation produces a mapping from *contexts* to *usages*. This is to handle the fact that the type of a let-binding may depend on the type of its surroundings, *e.g.* a polymorphically typed function. A context is thus a mapping from variables to types, and *usages* record for each let-binding the types used to instantiate it. The duplication algorithm proceeds as follows: when encoutering a let-binding we make one copy of it per usage of this binding. We proceed in a recursive fashion on the sub-expressions, choosing the current context according to the various usages we duplicate.

4.2.4 Partial CPS transformation

Once the program is transformed into simply-typed form, we can apply our partial CPS transformation, as defined earlier. But this CPS transformation generates too many administrative redexes, which may greatly hamper the performance of the resulting program. Take the example from the previous section. It is simply wrapped into an abstraction for pure expressions:

$$\begin{aligned} apply_{\mathcal{P}} &= \lambda f.\lambda x.fx \\ T_1[\![apply_{\mathcal{P}}]\!] &= \mathbf{ret}\ \ apply_{\mathcal{P}} \end{aligned}$$

Assuming that f and x are always applied to pure arguments, *i.e.* $flow(f) = flow(x) = flow(f\,x) = \mathcal{P}$, then we have this variation:

$$\begin{aligned} apply_{\mathcal{I}} &= \lambda f.\lambda x.\mathbf{let}\,() = \mathbf{yield\ in}\ fx \\ T_1[\![apply_{\mathcal{I}}]\!] &= \lambda k_0.k_0\lambda f.\lambda k_1.k_1\lambda x.\lambda k_2.(\lambda k_3.Waiting\ k_3)(\lambda().(\lambda().\lambda k_4.k_4(f\ x))\ ()\ k_2) \end{aligned}$$

There is still many administrative redexes, but pure expressions are preserved from being transformed. On real-world programs, most of the computation takes place in pure expressions, making the CPS part less of a burden. We thus defined an optimizing CPS transform which creates no administrative redex (adapted from [12, 13]) and which is equivalent to the initial one.

4.3 Polymorphic type inference

Monomorphisation operates on a typed source tree. To this end, we extended our type system to handle a caml-like language. Instead of modifying OCaml's type inference code, we chose to code from scratch a full-blown type inference system, handling let-polymorphism. Drawing upon [36], we decided to use a constraint-based inference algorithm. We use the (non-relaxed) value restriction to ensure the soundness of our analysis in presence of references.

As in [36], polymorphism is handled using constrained type schemes, whose meaning is roughly the set of all ground types admitted by the underlying expression. The constraint generation is defined inductively on expressions and is a quite natural encoding of the typing rules into the constraint language. This is no surprise since our type system is syntax-directed. It is also parameterised by the expected type of the expression. A formal definition of this algorithm can be found in [16].

4.4 Performances issues

For programs that use a small number of super-threads, this new implementation of the superposition does not improve the performances. This is mainly due to the fact that most of the computations are done within parallel vectors: replicated code is most of the time used to coordinate the works of the processors.

Thus expressions within vectors are pure and are not transformed: our type inference allows that. The performances should clearly improve whenever the number of super-threads is great. This is the subject of the next section.

5. Application to algorithmic skeletons

Anyone can observe that many parallel algorithms can be characterised and classified by their adherence to a small number of generic patterns of computation — farm, pipe, *etc.* Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit with specifications which transcend architectural variations but implementations which recognise them to enhance performance [11]. The core principle of skeletal programming is conceptually straightforward. Its simplicity is its strength.

A well know disadvantage of skeleton languages is that the only admitted parallelism is usually that of skeletons while many parallel applications are not obviously expressible as instances of skeletons. Skeletons languages must be constructed as to allow the integration of skeletal and ad-hoc parallelism in a well defined way [11].

In this light, having skeletons in BSML would have the advantage of the BSP pattern of communications (collective ones) and the expressivity of the skeleton approach. For our purposes and to have interesting benchmarks, we take for model the implementation of the OCamlP3L skeletons language (P3L's set of skeletons for OCaml) and base them on our parallel superposition primitive.

```
val seq : (unit → 'a → 'b) → unit → 'a stream → 'b stream
val loop:('a→ bool)∗(unit→ 'a stream→ 'a stream)→ unit→ 'a stream→ 'a stream
val farm : (unit → 'b stream → 'c stream) ∗ int → unit → 'b stream → 'c stream
val pipe (|||) : (unit → 'a stream → 'b stream) → (unit → 'b stream → 'c stream) → unit → 'a stream → 'c stream
val mapvector : (unit → 'b stream → 'c stream) ∗ int → unit → 'b array stream → 'c array stream
val reducevector : (unit → ('b ∗ 'b) stream → 'b stream) ∗ int → unit → 'b array stream → 'b stream
```

Figure 8. The types of the OCamlP3L skeletons

5.1 The OCamlP3L Skeletons

Fig. 8 [10] subsumes the ML type of the OCamlP3L skeletons. They work as follow.

The **seq** skeleton encapsulates an OCaml function f into a stream process which applies f to all the inputs received on the input stream and sends off the results on the output stream. **loop** computes a function f over all the elements of its input stream until a boolean condition g is verified.

The **farm** computes in parallel a function f over different data items appearing in its input stream. Parallelism is gained by having n independent processes.

The **pipeline** skeleton performs in parallel the computations relative to different stages of a function composition over different data items of the input stream.

mapvector computes in parallel a function over all the data items of a vector, generating the new vector of the results. The **reducevector** works in the same manner but doing an array folding with a binary operator as argument.

5.2 BSML Implementation

There are already some advantages to using BSML-based skeletons: BSML can be used on a wide variety of communication libraries, such as BSPlib, MPI and TCP/IP whereas OCamlP3l is currently stuck with TCP/IP.

We do not present how all skeletons and utility functions are implemented. We refer to [16] for more details.

5.2.1 Execution of process networks

The combination of P3L's skeletons generates a process network. This network takes in input a stream of data. Then each datum is transformed by the network independently of other data and finally the output is another stream of the same arity. In this way, these computations can be composed: supposing a n data stream, the execution of the network will be composed n times using the superposition.

For each execution of a network, we use a counter (place) that stores the placement of tasks and data in a round robin fashion. We then define a triplet which represent the network (input CPU, output CPU and the parallel stream node computation): 

where "node" is a function that takes a data from CPU "in" and return a data to CPU "out".

That will be implemented in BSML as a triplet where the P3L stream is implemented as a parallel vector of option values where only one processor keeps a non empty value — the data of the stream. The full stream is thus a list of these vectors. Now, to produce the process network we recursively generate a BSML code from a skeleton expression in meta fashion.

5.2.2 Implementation of the skeletons

The generated code of **seq**(f) is the network (pl,pl,(**fun** data → new_data)) where the function f only executes itself on the designated CPU pl (designated by the counter), returning None elsewhere.

For the **pipeline**(s_1,s_2) skeleton, we directly compose them: the input of the resulting network is the input of the network of s_1 whose output is the input of s_2.

If they are on distinct CPUs, we perform a sending function to connect the output of s_1 to the input of s_2.

Note that for a BSP machine with **p** processors and a pipe of two sequential processus, the tasks would be distributed on all processors — we suppose a typical stream of more than **p** elements. Then, a single barrier would occur sending data from a processor to another one. This is clearly not the most efficient manner to execute the whole program but this is also clearly not an inefficient one.

For the **farm**(n,s) skeleton, because we have a fixed number **p** of processors, we ignore the n parameter which represents the "number of workers". The parallelism degree is thus all the time **p**. In this way, the code generated for **farm**(n, s) is simply the code generated for the sub-skeleton s.

Our implementation of the **loop**(s,f) skeleton is a simple recursive function, which executes the s skeleton until the f condition holds true.

The **mapvector**(n,s) skeleton is probably the most interesting one. Once again, the parallelism degree n is unused. The method is as follow.

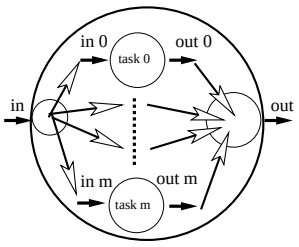
First, a new task is dynamically created for each element of the input vector of the stream and stored in a list of tasks, called ntasks. The BSML code for these tasks (produced by the sub-skeleton s) is generated from an inductive call.

Then, once all the tasks created, their execution are *superposed* using the superposition — **super_list** utility function. For each execution, the input processor of the network sends a data of the vector to the processor that has been dynamically designated to execute the sub-network. The parallelism arises from data being distributed over all superposed tasks.

Finally, we gather the results to the network output processor — **rebuild** utility function. This is exactly what is reflected in the code:

```
let pl=(!place) in (pl,pl,(fun data → let ntasks = ref [] in
 let size = noSome ((proj (applyat pl (fun t → Some (Array.length (noSome t))) (fun _ → Some 0) data)) pl) in
  for j=0 to (size−1) do
   incr_place ();
   let i,o,task= .<Code(s)>. in
   let new_task= (fun () → sendto o pl (task (sendto pl i (parfun
                    (function Some t→ Some t.(j) | None→ None) data)))) in
    ntasks:=new_task::(!ntasks);
  done;
  rebuild pl (super_list !ntasks)))
```

where .<Code(s)>. would be the BSML code of sub-skeleton s. This figure subsumes the idea of the implementation:



where $m+1$ is the size of the input vector. This skeleton is a good sample: the size of typical data (arrays) would make the past implementation (using system threads) of the superposition unusable in practice.

5.3 Benchmark

Our example is a parallel PDE solver which works on a set of subdomains, taken from [10]. On each subdomain it applies a fast Poisson solver written in C. The skeleton expression of the code is shown in Figure 9 and the coupling technique (and full equations) can be found in [10].

All the tests were run on the new LACL cluster composed of 20 Pentium dual core 2Ghz with 2GB of RAM interconnected with a Gigabyte Ethernet network.

We present the benchmarks when the interface meshes match using randomly generated sub-domains — real life inputs are described in [10]. The principle of this extensibility test is as follow: increase number of processors as well as size of data.

```
let PDE_solver = parfun (fun () →
 (loop ((fun (v,continue) → continue),seq(fun _ → fun (v,_) → v) ||| mapvector(seq(fun _→ compute_sub_domain),3)
    ||| seq(fun _→ projection) ||| seq(fun _→ bicgstab) ||| seq(fun _→ plot))))
```

Figure 9. Skeleton code fragment from a Poisson solver

In this context, for each input, one processor is associated with one sub-domain and the global domain is divided into 1, then into 2, 4, *etc.* sub-domains.

Various manners of decomposing the global domain in a structured way are explored. The number of sub-domains along the axis is denoted by N_x (resp. N_y, N_z) and each sub-domain possesses approximately 50000 cells — time to sequentially decompose a sub-domain is approximately linear. The number of generated super-threads would be too big for our past implementation.

Performances (minutes and seconds) of OCamlP3L and BSML (using its MPI implementation) are summarised in the following table:

(N_x, N_y, N_z)	Nb procs	OCamlP3L	BSML
$1 \times 1 \times 1$	1	20.56	21.29
$1 \times 1 \times 2$	2	24.06	27.63
$1 \times 1 \times 4$	4	24.78	28.23
$1 \times 1 \times 8$	8	25.05	28.97
$1 \times 1 \times 16$	16	26.53	30.67
$1 \times 2 \times 2$	4	20.78	25.14
$1 \times 2 \times 4$	8	24.45	28.36
$1 \times 2 \times 8$	16	25.56	29.84
$1 \times 4 \times 4$	16	26.89	29.89
$2 \times 2 \times 2$	8	25.88	27.21
$2 \times 2 \times 4$	16	27.89	32.75

As might be expected, OCamlP3L is faster than our naive implementation but not much. Barriers slow down the whole program but bulk-sending accelerates the communications: in the P3L run there exists a bottleneck due to the fact that sub-domains are centralised and therefore the amount of communication treated by one process may cause an important overhead. In BSML, the data are each time completely distributed, which reduces this overhead but causes a loss of time.

We did not benchmark the old implementation against the newer, because the older could not handle the number of concurrent threads. Our aim was not to beat OCamlP3L, whose implementation is far more complicated than ours but have both BSML and OCamlP3L and to not be stuck with TCP/IP as OCamlP3L is.

6. Related works

6.1 Divide-and-conquer and skeletons paradigms

A general data-parallel formulation for a class of divide-and-conquer problems was evaluated in [3]. But those techniques are only defined for a low-level parallel language, High Performance Fortran. In [25], the authors present a new data-parallel C library for Intel's core-processors which has a divide-and-conquer primitive. Some optimisations in the implementation have been done using the BSP model.

The approach of [29] distinguished two levels of abstraction: (1), a small skeleton language defines the static parallel parts of the programs; (2), an implementation of a divide-and-conquer skeleton demonstrates how meta-programming can generate the appropriate set of communications. However, no cost prediction nor efficient code generation are possible. For efficient code, [15] proposes using C++ templates but no divide-and-conquer skeleton is at this time provided.

[11] described how to add skeletons in MPI as well as some experiments — the eskel library. It also gives convincing and pragmatic arguments to mixed message

passing and skeleton programming, using C. We think that using OCaml for parallel programming (high-performance applications) is not a bad choice since the generated code is often very competitive with the C counterparts.

6.2 CPS transformations

The original CPS was the most simple one: it introduces too many unnecessary administrative redexes and more efficient CPS were defined latter in [12, 13]. CPS were massively used for various implementations of ML languages [2]. Historically, the idea of using CPS or a call-cc operator (call-with-current-continuation) for thread implementation comes from [45]. These techniques were then used to implement some concurrent extensions of sequential languages [5, 9, 41].

7. Conclusion and future works

7.1 Conclusion

In this paper we have defined a new implementation of a multi-threading primitive, called parallel superposition, for a high-level BSP and data-parallel language. This implementation uses a global CPS transformation which has been optimised using a flow analysis. Our presentation of the CPS transformation abstracts away from the details of BSP communications, as they are irrelevant to the semantics study. Different optimisations such as monomorphi(*flow*)sation have also been added for performance issues. Our implementation relies on semantics investigations, allowing us to better trust it. Furthermore, it works on an important subset of OCaml.

The presented techniques are not novel, except our CPS transformation and the monoflowisation. We would like to emphasise that this transformation is not a subset of the one in [31]. More precisely, we do not constrain the flow of a binding variable to be the same as its binding expression, allowing impure functions to take pure arguments. Moreover, we find that the combination of all our transformations on a large subset of OCaml is quite new, if not on the pure theoretical front, at least as a tool (we think that it could be adapted to handle other constructs such as call/cc) and it could be applied to many strict BSP high-level language such as ones of [7, 27, 30]. Also, we are not aware of any implementation of P3L skeletons using the pure BSP paradigm: they only use pre-existing low level libraries.

7.2 Future works

The ease of use of this new implementation of the superposition will be experimented by developing less naive implementations of the OCamlP3l's skeletons and using a smarter heuristic for load balancing computations which will depend of the BSP's architecture parameters. We will also investigate a realistic polyvariant flow analysis to generate less CPS code.

In [24], we present how to manage exceptions in BSML. The case where a local exception inside a parallel vector is not caught makes that one (or more) processor can execute a different replicated code than other ones, which is not allowed in our model of execution: replicated parts of the code are done by all processors else deadlocks can occur. Each time a local exception is not locally caught, we dynamically forbid the creation of parallel vectors and broadcast the exception to the other processors. In case of exceptions in two super-threads, we take into account only one of the exceptions [23]. This management is thus currently not

compatible with the current CPS transformation of the code. We think that just adding in the continuation the potential exception that was not caught inside a parallel vector is sufficient. We did not found any counter-example, but the proof has to be done using the semantics of [23].

References

[1] M. Alt. *Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance*. PhD thesis, Universität Münster, 2007.
[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
[3] M. Aumor, F. Arguello, J. Lopez, O. Plata, and L. Zapata. A data-parallel formulation for divide-and-conquer algorithms. *The Computer Journal*, 44(4):303–320, 2001.
[4] M. Bamha and M. Exbrayat. Pipelining a Skew-Insensitive Parallel Join Algorithm. *Parallel Processing Letters*, 13(3):317–328, 2003.
[5] E. Biagioni, K. Cline, P. Lee, C. Okasaki, and C. Stone. Safe-for-space threads in standard ml. *Higher-Order and Symbolic Computation*, 11(2):209–225, 1998.
[6] R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.
[7] O. Bonorden, J. Gehweiler, and F. Meyer auf der Heide. A Web Computing Environment for Parallel Algorithms in Java. *Scalable Computing: Practice and Experience*, 7(2):1–14, 2006.
[8] O. Bonorden, B. Juurlink, I. Von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.
[9] J. Chroboczek. Continuation Passing for C: A space-efficient implementation of concurrency. Technical report, PPS (University of Paris 7), 2005.
[10] F. Clment, V. Martin, A. Vodicka, R. Di Cosmo, and P. Weis. Domain Decomposition and Skeleton Programming with OCamlP3l. *Parallel Computing*, 32:539–550, 2006.
[11] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.
[12] O. Danvy and L. R. Nielsen. Cps transformation of beta-redexes. *Information Processing Letterseses*, 94(5):217–225, 2005.
[13] Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In *Logic for Programming, Artificial Intelligence and Reasoning, (LPAR)*, volume 4790 of *LNAI*, pages 211–225. Springer, 2007.
[14] F. Dehne. Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14:173–421, 1999.
[15] J. Falcou, J. Serot, T. Chateau, and J. T. Lapreste. QUAFF : Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32(7-8):604–615, 2006.
[16] I. Garnier and F. Gava. New Implementation of a Parallel Composition Primitive for a Functionnal BSP Language. Technical Report 5, LACL, University of Paris East, 2008.
[17] F. Gava. *Approches fonctionnelles de la programmation parallèle et des méta-ordinateurs ; Sémantiques, implantation et certification*. PhD thesis, University of Paris-East, 2005.
[18] F. Gava. External Memory in Bulk Synchronous Parallel ML. *Scalable Computing: Practice and Experience*, 6(4):43–70, 2005.
[19] F. Gava. Implementation of the Parallel Superposition in Bulk-Synchronous Parallel ML. In Y. Shi, G.D.v. Albada, J. Dongarra, and P.M.A. Sloot, editors, *The International Conference on Computational Science (ICCS), Part I*, volume 4487 of *LNCS*, pages 611–619. Springer-Verlag, 2007.
[20] F. Gava. A Modular Implementation of Parallel Data Structures in BSML. *Parallel Processing Letters*, 18(1):39–53, 2008.
[21] F. Gava. BSP Functional Programming; Examples of a cost based methodology. In M. Bubak et al., editor, *ICCS*, volume 5101 of *LNCS*, pages 375–385. Springer-Verlag, 2008.
[22] F. Gava and F. Loulergue. A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting. *Future Generation Computer Systems*, 21(5):665–671, 2005.
[23] L. Gesbert. *Développement systématique et sureté d'exécution en programmation parallèle structurée*. PhD thesis, University of Paris-East, 2009.
[24] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski. Bulk Synchronous Parallel ML with Exceptions. *Future Generation Computer Systems*, 2009. to appear.
[25] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-scale Architectures. Technical report, Intel Research, 2007.
[26] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM TOPLAS*, 26(1):47–56, 2004.
[27] Y. GU, B.-S. Le, and C. Wentong. Jbsp: a bsp programming library in java. *Journal of Parallel and Distributed Computing*, 61(8):1126–1142, 2001.
[28] N. Heintze. Control-Flow Analysis and Type Systems. In A. Mycroft, editor, *Static Analysis Symposium (SAS)*, number 983 in LNCS. Springer, 1995.
[29] C.A. Herrmann. Generating message-passing programs from abstract specifications by partial evaluation. *Parallel Processing Letters*, 15(3):305–320, 2005.
[30] K. Hinsen. Parallel scripting with Python. *Computing in Science & Engineering*, 9(6), 2007.
[31] J. Kim and K. Yi. Interconnecting between CPS terms and non-CPS terms. In A. Sabry, editor, *Third ACM SIGPLAN Workshop on Continuations, Technical Report*, number 545. Computer Science Department, Indiana University, 2001.
[32] E. A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006.
[33] F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In Peter M. A. Sloot et al., editor, *ICCS*, number 2659 in LNCS, pages 223–232. Springer, 2003.

[34] F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *ICCS*, number 2669 in LNCS. Springer Verlag, june 2003.
[35] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.
[36] F. Pottier and D. Rémy. The Essence of ML Type Inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
[37] O. Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Second ACM SIGPLAN Workshop on Continuations*, 1997.
[38] Julien Signoles. Calcul statique des applications de modules paramétrés. In *JFLA*, pages 21–36, 2003.
[39] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
[40] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.
[41] S. Srinivasan. A thread of one's own. In *New Horizons in Compilers Workshop*, 2006.
[42] H. Thielecke. From control effects to typed continuation passing. *ACM SIGPLAN Notices*, 38(1):139–149, 2003.
[43] A. Tiskin. A New Way to Divide and Conquer. *Parallel Processing Letters*, 11(4):409–422, 2001.
[44] P. Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 7(1):39–56, 1994.
[45] M. Wand. Continuation-based multiprocessing. In *Lisp Conference*, pages 19–28. ACM, 1980.