Implementation of the Parallel Superposition in Bulk-Synchronous Parallel ML

Frédéric Gava

Laboratory of Algorithms, Complexity and Logic, University of Paris XII
gava@univ-paris12.fr

Abstract. Bulk-Synchronous Parallel ML (BSML) is a functional data-parallel language to code Bulk-Synchronous Parallel (BSP) algorithms. It allows an estimation of execution time, avoids deadlocks and nondeterminism. This paper presents the implementation of a new primitive for BSML which can express divide-and-conquer algorithms.

Keywords: BSP Functional Programming, divide-and-conquer.

1 Introduction

Bulk-Synchronous Parallel ML (*BSML*) is an extension of ML to code *Bulk-Synchronous Parallel* (BSP) algorithms as functional programs in direct mode. BSP is a parallel programming model (we refer to [2,9] for an gentle introduction to BSP) which offers a high degree of abstraction and allows scalable and predictable performance on a wide variety of architectures with a realistic cost model based on a structured parallelism. Deadlocks and non-determinism are avoided. BSP programs are portable across many parallel architectures and BSML expresses them with a small set of primitives. These primitives are implemented as a parallel library (`http://bsmllib.free.fr`) for the functional programming language Objective Caml (`OCaml`). Using a safe high-level language as ML to programming BSP algorithms allows performance, scalability and expressivity.

The BSP model does not allow to synchronize a subset of the processors. This is often considered as an obstacle to express divide-and-conquer algorithms in the BSP model. Nevertheless it is showed in [9] that for typical applications, exploiting the loss of efficiency due to the lack of computation-communication overlapping is outweighed by the advantages of bulk data transfer, while making programming and debugging much more difficult.

Nevertheless, [10] argues that the divide-and-conquer paradigm fits naturally into the BSP model, without any need of subset synchronization. It proposes a method which is fully compliant with the BSP model. This method is based on sequentially interleaved threads of BSP computation, called *superthreads* (more explanation what superthreads are and how they support the divide-and-conquer scheme can be found in [10]). [7] presents a new primitive for BSML called *parallel superposition* and its associated cost model. This primitive is based on a notion similar to Tiskin's superthreads and was only informally described as equivalent to pairing. Adding this primitive in BSML allows to programming

bsp_p: unit→int	**bsp_l**: unit→float	**bsp_g**: unit→float
mkpar: (int→α)→α **par**		**apply**: (α →β)**par**→α **par**→β **par**
put: (int→α option)**par**→(int→α option)**par**		**proj**: α option **par**→int→α option
type α option = None | Some **of** α		**super**: (unit →α) →(unit →β) →α ∗ β

Fig. 1. The core `BSMLlib` library

more easily divide-and-conquer BSP algorithms. In the current paper we present the behavior of the parallel implementation.

2 Functional Bulk-Synchronous Parallel ML

BSML does not rely on SPMD (Single Program Multiple Data) programming. Programs are identical to usual sequential `OCaml` programs but work on a parallel data structure. Some of the advantages are better readability.

The `BSMLlib` library is based on the elements given in Figure 1. It gives access to the BSP parameters of the underling architecture. For example, **bsp_p**() is p, the *static* number of processes. There is an abstract polymorphic type α **par** which represents the type of p-wide parallel vectors of objects of type α one per processor. Those parallel vectors are created by **mkpar** so that (**mkpar** f) stores (f i) on process i for i between 0 and $p-1$: **mkpar** f $= \boxed{(\text{f } 0)}\boxed{\cdots}\boxed{(\text{f i})}\boxed{\cdots}\boxed{(\text{f } (p-1))}$

Asynchronous phases are programmed with **mkpar** and with **apply** such that (**apply** (**mkpar** f) (**mkpar** e)) stores ((f i)(e i)) on process i:

$$\textbf{apply}\ \boxed{\cdots}\boxed{f_i}\boxed{\cdots}\ \ \boxed{\cdots}\boxed{v_i}\boxed{\cdots} = \boxed{\cdots}\boxed{(f_i\ v_i)}\boxed{\cdots}$$

The **put** primitive expresses communication and synchronization phases. Consider the expression: **put**(**mkpar**(**fun** i→fs_i)) (∗). To send a value v (resp. no value) from process j to process i, the function fs_j at process j must be such as (fs_j i) evaluates to Some v (resp. None). The expression (∗) evaluates to a parallel vector containing functions fd_i of delivered messages. At process i, (fd_i j) evaluates to Some v (resp. None) if process j sent the value v (resp. no value) to the process i. The **proj** primitive also expresses communication and synchronization phases. (**proj** vec) returns a function f such that (f n) returns the nth value of the parallel vector vec. If this value is the empty value None then process n sends no message to the other processes. Otherwise this value is broadcast. Without this primitive, the global control cannot take into account data computed locally.

The primitive **super** effects *parallel superposition*, which allows the evaluation of two BSML expressions as interleaved threads of BSP computationsin. From the programmer's point of view, the functional semantics of the superposition is the same as pairing but of course the evaluation of **super** E_1 E_2 is different from the evaluation of (E_1, E_2) [4]). 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 of E_2. The messages are obtained by concatenation of the messages and only one barrier occurs. If the evaluation of E_1 needs more supersteps than that of E_2 then the evaluation of E_1 continues (and *vice versa*).

3 Implementation

The implementation of the superthreads needed for the superposition uses the thread feature of OCaml. Each superthread is defined as a thread associated with an identifier and a channel of communication (à la Concurrent ML) to sleep or wake up a superthread. A specific scheduler is thus needed. We also need an environment of communication defined as a hash table (where keys are the identifier of the superthreads).

3.1 New Implementation of the Primitive of BSML

In this new implementation of the BSMLlib library, the core module which contains the primitives presented in section 2, is implemented in SPMD style using a lower level communication library, a module for the scheduling of the superthreads called Scheduler, a module for the environment of communication called EnvComm and a module of generic operators for the parallel superposition called SuperThread. The implementation of all the other modules of the BSMLlib library is independent of the actual implementation of these modules. The module of communication called Comm is based on the following elements [8]:

```
pid: unit→int        nprocs: unit→int        send: α option array→α option array
```

There are several implementations of Comm based on MPI (which used the MPI MPI_Alltoall C operator), PUB [3] and TCP/IP (only using the TCP/IP features of OCaml). The meaning of pid and nprocs is obvious. The function Comm.send takes on each process an array of size nprocs() of optional values. If at process j the value contained at index i is (Some v) then the value v will be sent from process j to process i. If the value is the None value, nothing will be sent. The result is an array of sending values. A global synchronization occurs inside this communication function.

The implementation of the abstract parallel vectors, **mkpar** and **apply** is as follows (rules 7.2 and 7.3, page 121 of the small-steps semantics in [4]):

```
type α par = α      let mkpar f = f (Comm.pid())      let apply f v = (f v)
```

The communication primitives of the BSMLlib, i.e, the **put** and **proj** primitives are also implemented as in the small-steps semantics (rules 7.4 and 7.5):

```
let send v = let id=(Scheduler.pid_superthread_run SuperThread.our_schedule) in
  EnvComm.add SuperThread.envComm id v; (SuperThread.rcv())
let mkfuns = (fun res i →if ((0<=i)&&(i<(!nprocs))) then res.(i) else None)
let put f = mkfuns (send (Array.init (!nprocs) f))
let proj v = put (fun _ →v)
let super f1 f2 = let t=(SuperThread.create_child f2) and v=f1 ()
                  in (v, SuperThread.wait t)
```

The send operator (rule 7.8) takes the identifier (Scheduler.pid_superthread_run) of the current active superthread (from the scheduler SuperThread.our_schedule) and puts the values to send in the environment of communication (EnvComm). Then it returns the received values after the communication phase.

The primitive **super** is also implemented as in the semantics (rule 7.10 page 122), i.e., we build a pair where in the first component, we compute f1 and in the

```
module SuperThread:functor(HowComm:sig val make_comm:(unit→unit) ref end)→sig
 val envComm : unit EnvComm.t                    val our_schedule : Scheduler.t
 val rcv : unit →α                               type β data_of_thread
 val create_child: (unit→β )→β data_of_thread val wait: β data_of_thread →β
end
```

Fig. 2. The Super Module

second component we run a child as another superthread to compute f2 and we wait for its result. SuperThread.create_child run another superthread and return the identifier of the new superthread which is the argument of the **wait** operator.

In [4], we describe how we implemented the module Scheduler which helps to schedule the superthreads (with the strategy of the small-steps semantics) and the module EnvComm of the environment of communication.

3.2 Functions for the Implementation of the Superposition

The module for implementing the parallel superposition called SuperThread is a functor based on the elements given in Figure 2. We have an environment of communication called envComm, a scheduler called our_schedule, the abstract type β data_of_thread of a superthread and the functions used above.

This functor is parameterized by a module which contains a reference to a function that makes the communication. This reference would be affected at the initialization with a function that would manipulate the environment of communication (iterating the hash-table) and perform the communications using the function send of the module Comm. Now, we will describe the implementation of the principal elements of this module.

The first one is the rcv operator which works as in rule 7.9 of the small-steps semantics i.e., this operator returns and deletes from the environment of communication the values received by the superthread at the current superstep. rcv is implemented using the functions of the scheduler which works as follow. We first test if the current superthread is the last superthread of the superstep. If it is the case, communications are done because we are at the end of the superstep. Then, we test if the current superthread is not the only one to run. If it is the case, those superthreads need to be run (strategy of the semantics: after the blocked communications, superthreads need to continue their works) and thus, we sleep the current superthread and wake up the first of them. Note that in this case the scheduler would give the hand to the current active superthread in the future. If not i.e., the current superthread is the only one, we finish by returning the value read from the environment. of communication.

The second one is the wait operator which returns the final result of the superthread child. It works as follow. First we test if the child has finished its computation. If not the father has to wait this result and thus we remove the superthread from the scheduler. We also wake up the next superthread and make sleep the current superthread. Its child would wake up it in the future. To finish, the result of the child is returning.

The last one, is the creation of a child (a new superthread) with the create_child operator which works as follow. First, a new superthread is created as a new `OCaml`'s thread which first sleeps (strategy of the semantics), computes the final result and to finish. The end of the child works as follow. First, we test if the father has ended its computation or not. If it is the case, then the child wakes up its father. Else, the superthread is removing from the scheduler and the next superthread is waked up. We also test if the superthread child is the last superthread of the superstep. If it is the case, communications are done because we are at the end of the superstep.

4 Example and Benchmarks

4.1 Calculus of the Prefix Using the Parallel Superposition

The example presented below is a divide-and-conquer version of the scan program using the parallel superposition. In this version of the calculus of the prefix, the processors are divided into two parts and the scan is recursively applied to those parts (Figure 3). 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 operator op on the second part.

In our benchmarks, we will make a performance comparison of the divide-and-conquer version of the computation of the prefix with two other versions. The first one, is the direct version and the second one is the binary computation using $\log(p)$ supersteps [2] those coded in Figure 4.

```
let inbounds first last n = (n>=first)&&(n<=last) (* inbounds: α →α →α →bool *)
let within_bounds = inbound 0 (bsp_p()−1) (* mix: int→α par * α par→α par *)
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 replicate e = mkpar (fun _ →e) (* replicate: α →α par *)
let parfun f v = apply (replicate f) (* parfun:(α →β )→α par→β par *)
let parfun2 f v1 v2 = apply (parfun f v1) v2

(* scan: (α →α →α )→α →α par→α par *)
let scan_super 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')) (replicate mid)) vec' in
 scan' 0 (bsp_p()−1) op vec
```

Fig. 3. Code of the divide-and-conquer algorithm of the parallel prefix

```
let scan_direct op e vv =
 let mkmsg pid v dst=if dst<pid then None else Some v in
 let procs_lists=mkpar(fun pid→from_to 0 pid) in
 let rcv_msgs=put(apply(mkpar mkmsg) vv) in
 let values_lists= parfun2 List.map (parfun (compose noSome) rcv_msgs) procs_lists in
  applyat 0 (fun _ →e) (List.fold_left op e) values_lists

let scan_logp op vec =
 let rec scan_aux n vec =
  if n >= (bsp_p()) then vec else
   let msg = mkpar(fun pid v dst→
     if ((dst=pid+n)or(pid mod (2*n)=0))&&(within_bounds (dst−n))
      then Some v else None)
   and senders = mkpar(fun pid→natmod (pid−n) (bsp_p()))
   and op' = fun x y→match y with Some y'→op y' x | None →x in
    let vec' = apply (put(apply msg vec)) senders in
    let vec''= parfun2 op' vec vec' in
     scan_aux (n*2) vec'' in
 scan_aux 1 vec
```

Fig. 4. Code of the direct and binary algorithm of the parallel prefix

4.2 Benchmarks

We did some preliminary experiments on a cluster with 10 Pentium IV nodes (with 1 Go of main memory per node) interconnected with a Gigabit Ethernet network. Both need $\log p$ supersteps except the direct version which need one superstep. The values were arrays of floats representing polynomials. The binary operation is the sum of two polynomials.

The MPI and TCP/IP implementations of the Comm module were used. These programs ran 100 consecutively times with initial randomized polynomials and this 5 times. The native code compiler of `OCaml` was used. In Figures 5, diagrams show the average of the results with increasing size of polynomials. In (a) and (c) (resp. (b) and (d)), we give the performances using the MPI (resp. TCP/IP) implementation of BSML.

The direct version is the faster for small polynomials. However, the version using the superposition and TCP/IP seems to be the faster one for big polynomials. For all the versions, the scalability of the BSP model is well-preserved.

We have also perform some performance comparisons of the direct and binary versions of the scan with a `BSMLlib` which does not contain the superposition and with our new one which supports superposition. We have show that the overhead, for programs which do not use superposition, is negligible.

5 Related works

The superthread way to divide-and-conquer in the framework of an object-oriented language was presented in [10]. There is no formal semantics and no

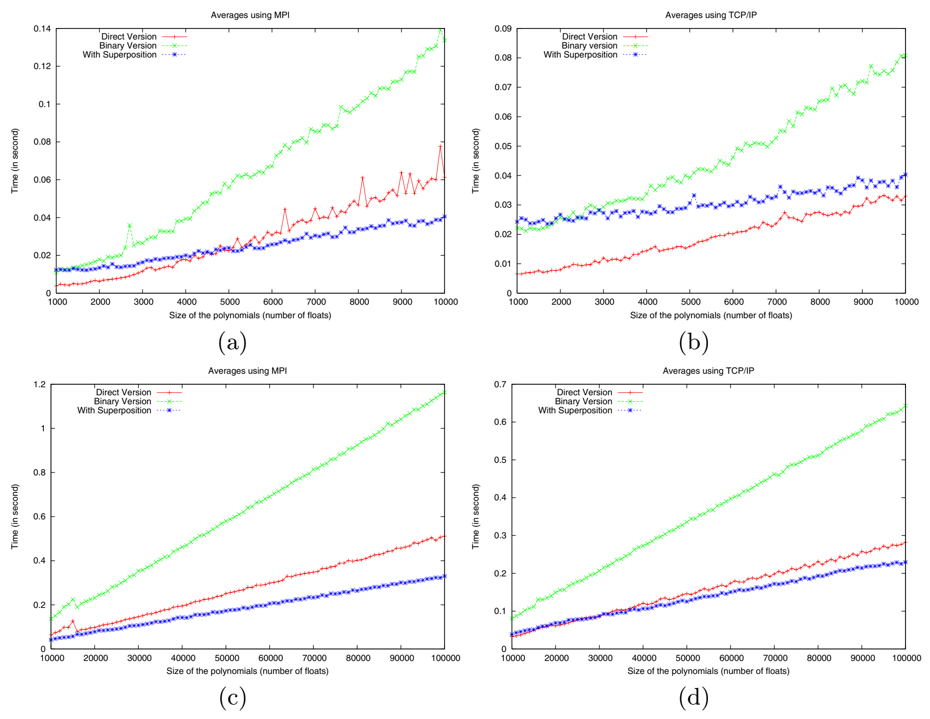

Fig. 5. Experiments of parallel prefix of polynomial using MPI or TCP/IP

implementation from now on. An algorithmic skeletons language which offers divide-and-conquer skeletons was designed in [12]. Nevertheless, the cost model is not really the BSP model but the D-BSP model which allows subset synchronization. We follow [9] to reject such a possibility.

A general data-parallel formulation for a class of divide-and-conquer problems was evaluated in [1]. A combination of techniques are used to reorganize the algorithmic data-flow, providing great flexibility to efficiently exploit data locality and to reduce communications. But those techniques are only define for a low-level parallel language, High Performance Fortran.

In [5], the proposed approach distinguished three levels of abstraction and their instantiations. (1), a small language, as an extension of ML, defines the static parallel parts of the programs. The language comes with a partial evaluator which acts as a code transformer using MetaOCaml. (2), an implementation of a divide-and-conquer skeleton demonstrates how meta-programming can generate the appropriate set of communications for a particular process from an abstract specification. (3), the application programmer composes the program using skeletons, without then need to consider details of parallelism. However, cost prediction nor native (efficient) code generation are possible.

6 Conclusion

The parallel superposition is a new primitive of BSML and it allows divide-and-conquer algorithms to be expressed easily, without breaking the BSP execution model. Compared to the parallel juxtaposition [6], this new primitive has not the drawbacks of its predecessor: the cost model is a compositional one and it can be seen as a purely functional primitive. We have presented in this paper how implements this new primitive with the help of the low-level semantics and makes some benchmarks of a classical BSP algorithms.

The ease of use of the superposition will be experimented by implementing BSP algorithms described as divide-and-conquer algorithms in the literature. An implementation of the superthreads using fault tolerant threads of MPI and static cost analysis as in [11] are also another directions of research.

Acknowledgments. This work is supported by the ACI Jeunes Chercheurs program from the French Ministry of Research, under the project PROPAC `wwwpropac.free.fr`. The author wishes to thanks Anne Benoît and Julien Signoles for their comments.

References

1. 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.
2. R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.
3. O. Bonorden, B. Juurlink, I. Von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.
4. F. Gava. *Approches fonctionnelles de la programmation parallèle et des méta-ordinateurs ; Sémantiques, implantation et certification.* PhD thesis, University of Paris XII, 2005.
5. C. A. Herrmann. Functional meta-programming in the construction of parallel programs. *Parallel Processing Letters*, 2006. to appear.
6. F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. In H. Kosch, L. Boszorményi, and H. Hellwagner, editors, *Euro-Par 2003*, number 2790 in LNCS, pages 781–788. Springer Verlag, 2003.
7. F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *International Conference on Computational Science (ICCS 2003), Part III*, number 2659 in LNCS, pages 223–232. Springer Verlag, june 2003.
8. F. Loulergue, F. Gava, and D. Billiet. Bulk-Synchronous Parallel ML: Modular Implementation and Performance Prediction. In V. S. Sunderem, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS), Part II*, number 3515 in LNCS, pages 1046–1054. Springer, 2005.
9. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

10. A. Tiskin. A New Way to Divide and Conquer. *Parallel Processing Letters*, 11(4):409–422, 2001.
11. P. B. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL'02*, LNCS, pages 110–125. Springer Verlag, 2003.
12. A. Zavanella. *Skeletons and BSP : Performance Portability for Parallel Programming*. PhD thesis, Universita degli studi di Pisa, 1999.