BSP Functional Programming: Examples of a Cost Based Methodology

Frdric Gava

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

Abstract. Bulk-Synchronous Parallel ML (BSML) is a functional data-parallel language for the implementation of Bulk-Synchronous Parallel (BSP) algorithms. It makes an estimation of the execution time (cost) possible. This paper presents some general examples of BSML programs and a comparison of their predicted costs with the measured execution time on a parallel machine.

Key words: BSP Functional Programming, Cost Prediction.

1 Introduction

Solving a problem is often a complex job especially when a parallel machine is used: it is necessary to manage communication, synchronisation, partition of data *etc.* at the same time. Algorithmic models and high-level languages are needed to simplify both the design of parallel algorithms (ability to compare their costs[1]) and their programmation in a safe, efficient and portable manner.

BSML is an extension of ML designed for the implementation of BSP algorithms as functional programs using a small set of parallel primitives. BSP [3,17] is a parallel model 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 small set of machine parameters. Deadlocks and non-determinism are avoided. BSML is implemented as a parallel library[2] for the functional programming language Objective Caml (`OCaml`).

Our methodology is as follow: first, analyse the complexity of the sequential algorithm, then design one or more parallel algorithms, analyse their BSP costs, calculate the BSP parameters of the parallel machines, program these algorithms in BSML and finally test the performance of the programs on different architectures. Using safe high-level languages like ML to program BSP algorithms (that is BSML) allows performance, scalability and expressivity.

Other approaches to safe high-performance computation exist. We can cite concurrent programming [6], more or less synchronous processes [5,18] or the automatic parallelization of programs [1] and algorithmic skeletons [7]. In the first two cases, the expressivity of concurrence or mobility is sought (but with lower

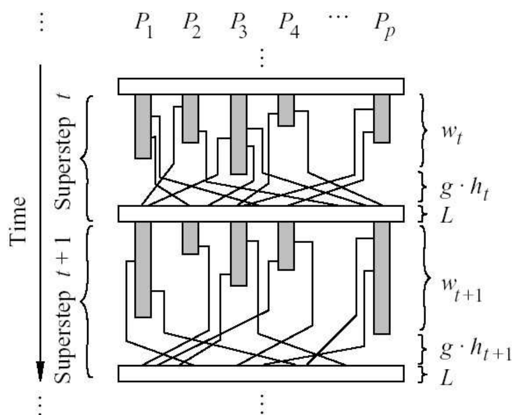
[1] We speak about complexity for sequential algorithms and cost for parallel ones.
[2] Web page at `http://bsmllib.free.fr`.

performance). In other cases, it's the simplicity of parallelism (which becomes almost transparent) which is sought. As opposed to these methods, we prefer the use of a performance model[3] at the cost of expressiveness.

Indeed, other approaches do not allow, in their intrinsic models, to predict the run-time[4]. That makes algorithmic optimisations and the choice of the best algorithm for a given architecture difficult: even if the number of *processes/threads/workers* or their computations can be limited, it is often difficult to analyse the communication times or the placement of computation and data on the processors. Nevertheless, it is possible to find empirical optimisations (by successive tests) or to design better schedulers [2]. But algorithmic optimisations are still hard to analyse. In this article, we illustrate our methodology with simple examples of problems that illustrate many aspects of classical algorithmic problems.

2 Functional Bulk-Synchronous Parallel Programming

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 inter-processor delivery of messages (for sake of conciseness, we refer to [3,17] for more details). A BSP program is executed as a sequence of *super-steps*, each one divided into three successive disjoint phases: each processor uses its local data (only) 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 performance of the BSP machine is characterised by 4 parameters: the local processing speed $\mathbf{r}$; the number of processor $\mathbf{p}$; the time $\mathbf{l}$ required for a barrier; and the time $\mathbf{g}$ 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.

[3] Note that this observation has already been made in the context of programming C+BSP matrix computations [12,14].

[4] Some tools [8,11] exist but are too much complex to be used or are not implemented.

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) **par**→(int→α) **par** **proj**: α **par**→int→α
super: (unit→α)→(unit→β)→$\alpha * \beta$

Fig. 1. The BSML primitives.

2.2 Bulk-Synchronous Parallel ML

BSML is based on 8 primitives (Fig. 1), three of which are used to access the parameters of the machine. Implementation of these primitives rely either on MPI, PUB [4] or on the TCP/IP functions provided by the Unix module of `OCaml`. A BSML program is built as a sequential program on a parallel data structure called parallel vector. Its ML type is α **par**, which expresses that it contains a value of type α at each of the **p** processors.

The BSP asynchronous phase is programmed using the two primitives **mkpar** and **apply** so that (**mkpar** f) stores (f i) on process i (f is a sequential function): **mkpar** f = | (f 0) | $\cdots$ | (f i) | $\cdots$ | (f (**p**−1)) | and **apply** applies a parallel vector of functions to a parallel vector of arguments: **apply** | $\cdots$ | f_i | $\cdots$ | | $\cdots$ | v_i | $\cdots$ | = | $\cdots$ | (f_i v_i) | $\cdots$ |

The first communication primitive is **put**. It takes as argument a parallel vector of functions which should return, when applied to i, the value to be sent to processor i. **put** returns a parallel vector with the vector of received values: at each processor these values are stored in a function which takes as argument a processor identifier and returns the value sent by this processor.

The second communication primitive **proj** is such that (**proj** vec) returns a function f where (f n) returns the nth value of the parallel vector vec. Without this primitive, the global control cannot take into account data computed locally.

The primitive **super** allows the evaluation of two BSML expressions as interleaved threads of BSP computations. From the programmer's point of view, the semantics of the superposition is the same as pairing but the evaluation of **super** E_1 E_2 is different: 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 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*).

2.3 Often Used Parallel Functions

The primitives described in the previous section constitute the core of the BSML language. In this section, we define some useful functions which are parts of the standard BSML library. For the sake of conciseness, their full code is omitted.

replicate:α →α **par** creates a parallel vector which contains the same value everywhere. The primitive **apply** can be used only for a parallel vector of functions that take one argument. To deal with functions that take two arguments we need to define the apply2:(α →β →γ)→α **par**→β **par**→γ **par** function.

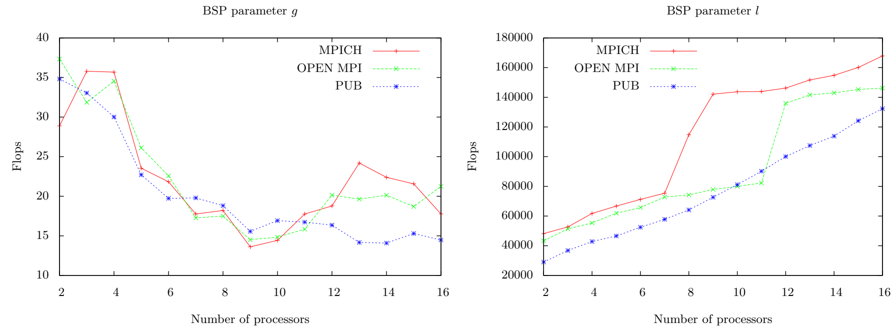


Fig. 2. BSP parameters of our machine.

It is also common to apply the same sequential function at each processor. We use the parfun functions where only the number of arguments to apply differs: parfun:$(\alpha \rightarrow \beta) \rightarrow \alpha$ **par**$\rightarrow \beta$ **par** and parfun2:$(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha$ **par**$\rightarrow \beta$ **par**$\rightarrow \gamma$ **par**.

It is common to perform a total exchange. Each processor contains a value (represented as a vector of values) and the result of (rpl_total:α **par**$\rightarrow \alpha$ list $\boxed{v_0}\boxed{\cdots}\boxed{v_{p-1}}$) is $[v_0, \ldots, v_{p-1}]$, a list of these values on each processor.

2.4 Computation of the BSP parameters

One of the main advantages of the BSP model is its cost model: it is quite simple and yet accurate. We used the BSML implementation [15] of a program [3] that benchmarks and determines the BSP parameters of our machine (16 Pentium IV 2.8 Ghz, 1 Gb RAM nodes cluster interconnected with a Gigabyte Ethernet network). We then compute these parameters for 3 libraries, corresponding to 2 different implementations of BSML: MPICH, OPEN-MPI[5] and PUB [4].

Fig. 2 summarises the timings (where **r** is 330 Mflops/s for each library) for an increasing number of processors. We notice that the parameter **l** is growing in a quasi-linear way for the library PUB. However, for the libraries MPI, two jumps are visible. No explanation has been found yet. For parameter **g**, it is surprising to see that it is high for a few processors and then more stable (real parameter exchange of the network). This is certainly due to the buffer management in communication protocols and OS: when the number of processors increases, the buffers are filled faster and messages are transmitted immediately.

3 Examples of BSP problems in BSML

To illustrate our methodology, we present 2 classic problems: sieve of Eratosthenes and N-body computing. For each problem, we give the parallel methods, BSP cost formulas as well as tests of comparison between theoretical (depending

[5] `http://www.mcs.anl.gov/mpi/mpich1` and `http://www.open-mpi.org/`

on the BSP parameters) and experimental performances. This comparison shows that the BSP cost analysis would help choosing the best BSML program.

3.1 Sieve of Eratosthenes

The sieve of Eratosthenes generates a list of primary numbers below a given integer n. We study 3 parallelization methods. We generate only the integers that are not multiple of the 4 first prime numbers and we classically iterate only to $\sqrt{n}$. The probability of a number a to be a prime number is $\frac{1}{\log(a)}$, so we deduce a complexity of $(\sqrt{n} \times n)/\log(n)$.

Fig. 3 gives the BSML code of the 3 methods. We used the following functions: elim:int list→int→int list which deletes from a list all the integers multiple of the given parameter; final_elim:int list→int list→int list iterates elim; seq_generate:int→int→int list which returns the list of integers between 2 bounds; and select:int→int list→int list which gives the $\sqrt{n}$th first prime numbers of a list.

Logarithmic reduce method For our first method we use the classical parallel prefix computation (also call folding reduce) :

$$\text{scan} \oplus \boxed{v_0 \mid \cdots \mid v_{p-1}} = \boxed{v_0 \mid v_0 \oplus v_1 \mid \cdots \mid \oplus_{k=0}^{p-1} v_k}$$

We use a divide-and-conquer BSP algorithm (implemented using the **super** primitive) 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 broadcasted 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. In our computation, the sent values are first modified by a given function (select to just sent the $\sqrt{n}$th first prime numbers)

The parallel methods is thus very simple: each processor i holds the integers between $i \times \frac{n}{\mathbf{p}} + 1$ and $(i+1) \times \frac{n}{\mathbf{p}}$. Each processor computes a local sieve (the processor 0 contains thus the first prime numbers) and then our scan is applied. We then eliminate on processor i the integers that are multiple of integers of processors $i-1$, $i-2$, *etc.* We have $\log(\mathbf{p})$ super-steps where each processor sent/received at most 2 values (list of size max $\sqrt{n}$). The BSP cost is accordingly:

$$\log(\mathbf{p}) \times (\frac{\sqrt{m} \times m}{\log(m)} + 2 \times \sqrt{n} \times \mathbf{g} + \mathbf{l}) \text{ where } m = \frac{n}{\mathbf{p}}.$$

Direct method It is easy to see that our initial distribution (bloc of integers) gives a bad load balancing (processor $\mathbf{p}-1$ has the bigger integers which have little probability to be prime). We will distributes integers in a cyclic way: a is given to processor i where $a \bmod \mathbf{p} = i$). The second method works as follows: each processor computes a local sieve; then integers that are less to $\sqrt{n}$ are globally exchanged; a new sieve is applied to this list of integers (thus giving prime numbers) and each processor eliminates, in its own list, integers that are multiples of this $\sqrt{n}$th first primes. The BSP cost is accordingly:

$$2 \times \frac{\sqrt{m} \times m}{\log(m)} + \frac{\sqrt{\sqrt{n}} \times \sqrt{n}}{\log(\sqrt{n})} + \sqrt{n} \times \mathbf{g} + \mathbf{l}$$

```
let eratosthene_scan n =
  let p=bsp_p() in
  let listes = mkpar (fun pid→ if pid=0 then seq_generate (n/p) 10
                               else seq_generate ((pid+1)*(n/p)) (pid*(n/p)+1)) in
  let local_eras = parfun (local_eratosthene n) listes in
  let scan_era = scan_super final_elim (select n) local_eras in
   applyat 0 (fun l →2::3::5::7::l) (fun l→l) scan_era

let eratosthene_direct n =
 let listes = mkpar (fun pid→ local_generation n pid) in
 let etape1 = parfun (local_eratosthene n) listes in
 let selects = parfun (select n) etape1 in
 let echanges = replicate_total_exchange selects in
 let premiers = local_eratosthene n
                              (List.fold_left (List.merge compare) [] echanges) in
 let etape2 = parfun (final_elim premiers) etape1 in
  applyat 0 (fun l→2::3::5::7::(premiers@l)) (fun l→l) etape2

let rec eratosthene n =
  if (fin_recursion n) then apply (mkpar distribution) (replicate (seq_eratosthene n))
  else
   let carre_n = int_of_float (sqrt (float_of_int n)) in
   let prems_distr = eratosthene carre_n in
   let listes = mkpar (fun pid →local_generation2 n carre_n pid) in
   let echanges = replicate_total_exchange prems_distr in
    let prems = (List.fold_left (List.merge compare) [] echanges) in
     parfun (final_elim prems) listes
let eratosthene_rec n =
 applyat 0 (fun l→2::3::5::7::l) (fun l→l) (eratosthene n)
```

Fig. 3. BSML code of the the parallel versions of the sieve of Eratosthenes.

Recursive method Our last method is based on the generation of the $\sqrt{n}$th first primes and elimination of the multiples of this list of integers. We generate this by a inductive function on n. We suppose that the inductive step gives the $\sqrt{n}$th first primes and we perform a total exchange on them to eliminates the non-primes. End of this induction comes from the BSP cost: we end when n is small enough so that the sequential methods is faster than the parallel one. The inductive BSP cost is accordingly:

$$\begin{array}{ll} \mathrm{Cost}(n) = \frac{\sqrt{m}\times m}{\log(m)} + \sqrt{n} \times \mathbf{g} + \mathbf{l} + \mathrm{Cost}(\sqrt{n}) & \\ \mathrm{Cost}(n) = \frac{\sqrt{n}\times n}{\log(n)} & \text{if BSP cost ¿ complexity} \end{array}$$

Fig. 4 gives the predicted and measured performances (using the PUB implementation). To simplify our prediction, we suppose that pattern-matching and modulo are constants in time. Size of lists of integers can be measured using the Marshal module of `OCaml`. Note that we obtain a super-linear acceleration for

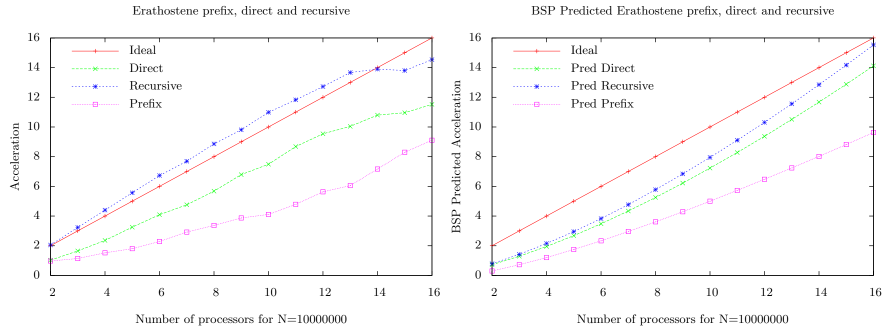


Fig. 4. Performances (using PUB) of the sieve of Eratosthenes.

the recursive method. This is due to the fact that, using a parallel method, each processor has a smaller list of integers and thus the garbage collector of `OCaml` is called less often. One can notice that predicted performances using the BSP cost model are close to the measured ones.

3.2 The N-body problem

The classic N-body problem is to calculate the gravitational energy of N point masses, which is given by:

$$E = -\sum_{\substack{i=1 \\ i \neq j}}^{N} \sum_{j=1}^{N} \frac{m_i \times m_j}{r_i - r_j}$$

The complexity of this problem is thus in order of magnitude of N^2. To compute this sum, we show two parallel algorithms : using a total exchange of the point masses or using a systolic loop[6]. At the beginning of these two methods, each processor contains a sub-part (as a list) of the N point masses: we thus have a parallel vector of lists of $N/\mathbf{p}$ point masses.

Fig. 5 gives the BSML code of the 2 algorithms. pair_energy computes the interaction of a list of masses with another one. The sequential method is thus a call of this function to the same list.

Total exchange method The method is naive: a total exchange of these lists is done and then processors compute the interaction of its own list with other ones; at the end, a parallel fold is applied to sum the partial interactions. The BSP cost is accordingly: $N \times g + 2 \times N + \frac{N}{p} \times N + l + p \times g + l$ that is two super-steps: time of the total exchange and the concatenation of the received lists; time to perform the local interactions and time to finish the fold.

[6] There exist more sophisticated algorithms that take advantage of the symmetry of the sum but this is not the subject of this article.

Systolic loop Our second algorithm is based on a systolic loop [13]. In such an algorithm, data is passed around from processor to processor in a sequence of super-steps. We can easily write a generic systolic loop in BSML:

```
(* val systolic:(α →α →β ) →(γ →β par →γ ) →α par →γ →γ *)
let systolic f op vec init =
 let rec calc n v res =
  if n=0 then res else
   let newv=Bsmlcomm.shift_right v in
    calc (n−1) newv (op res (Bsmlbase.parfun2 f vec newv))
  in calc (bsp_p()) vec init
```

with shift_right:α **par**→α **par** which shifts the values from each processor to its right-hand neighbour (part of the standard BSML library).

Initially, each processor receives its share of the N point masses and calculates the interactions among them. Then it sends a copy of its particles to its right-hand neighbour, while at the same time receiving the particles from its left-hand neighbour. It calculates the interactions between its own particles and those that just came in, and then it passes on the particles that came from the left-hand neighbour to the right-hand neighbour. After $\mathbf{p} - 1$ super-steps, all pairs of particles have been treated and a folding of these values can be done to finish the computation. The BSP cost is accordingly:

$$\mathbf{p} \times (\frac{N}{\mathbf{p}} \times \mathbf{g} + \mathbf{l} + 2 \times \frac{N}{\mathbf{p}} + \frac{N}{\mathbf{p}} \times \frac{N}{\mathbf{p}}) + \mathbf{p} \times \mathbf{g} + \mathbf{l}$$
$$\equiv N \times \mathbf{g} + \mathbf{p} \times \mathbf{l} + 2 \times N + \frac{N}{\mathbf{p}} \times N + \mathbf{l} + \mathbf{p} \times \mathbf{g} + \mathbf{l}$$

that is the same as before but with more synchronization time.

Fig. 6 gives the predicted and measured performance (using MPICH). Size of lists of particules are measured as before. One can notice that performances scales well. The naive method has better theoretical and practical performances than the systolic ones. The asset of the systolic method appears when the number of particules is so big that lists do not fit in the main memory of a node of the parallel machine: performance degenerates due to the paging mechanism used to get enough virtual memory. This is a limitation of the BSP model that could be solved using a more sophisticated one for *out-of-core* applications [9].

One can also notice that for our two examples (sieve of Eratosthenes and N-body), measured performances are sometime better than predicted ones. This is due to the fact that in some cases, communications can perform better than predicted ones ($\mathbf{g}$ and $\mathbf{l}$ are averages of network parameters).

4 Conclusion

BSML is a language for programming BSP algorithms. We have attempted to show that it is possible to predict the performance of BSP algorithms following the parameters of a given machine and so to choose what the most efficient and scalable BSML program is. We have illustrated this with two classical problems. Our work illustrates the importance of a high-level parallel paradigm with more compact and therefore more readable code without too bad performances.

```
type point = float * float * float and atom = point * float
let minus_point (x1,y1,z1) (x2,y2,z2) = (x1−.x2,y1−.y2,z1−.z2)
let length_point (x,y,z) = sqrt(x*.x +. y*.y+. z*.z)

(* val pair_energy : atom list →atom list →float *)
let pair_energy some_bodies other_bodies =
 List.fold_left (fun energy →function (r1,m1) →
     energy+.(List.fold_left (fun energy →function (r2,m2) →
                                          let r=length_point(minus_point r2 r1) in
                                           if r>0. then energy+.(m1*.m2)/.r else energy)
                                        0. other_bodies)) 0. some_bodies
(* Total exchange method *)
 let final_ex = parfun2 pair_energy my_bodies
                         (parfun List.concat (total_exchange my_bodies)) in
 let res_final= fold_direct (+.) 0. final_ex in
 ...
(* Systolic method *)
 let energy=parfun2 pair_energy my_bodies my_bodies in
 let final_sys = systolic pair_energy (parfun2 (+.)) my_bodies energy in
 let res_final= fold_direct (+.) 0. final_sys in
 ...
```

Fig. 5. BSML code of the parallel versions of the N-body problem.

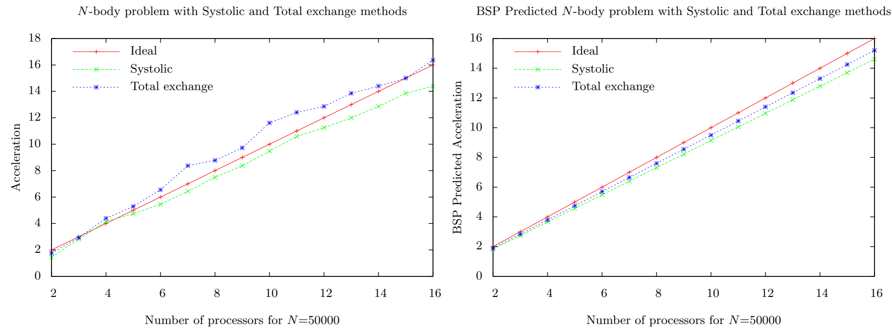


Fig. 6. Performance (using MPICH) of the N-body problem.

Even if our methodology might seem lengthy, we believe it is necessary for the future of parallel programming especially as multi-cores machines became the norm. Their programming (as well as clusters) in a safe, expressive, predictable and efficient manner will surely become one of the keys to software design.

Future work will naturally be comparison with other parallel languages and libraries as OCamlP3L, C+BSPlib, C+MPI, Eden or Gph [10] (and with bigger programs and other kinds of architectures as multi-cores ones) in order to validate our approach. Finally, manual cost analysis for functional programs has

its limits: it is necessary to estimate (sometimes by testing) the number of flops needed to make a pattern-matching, build a tuple, *etc.* We could use [16] in order to estimate them automatically.

Acknowledgement: Thanks to Louis Gesbert for its speel checking.

References

1. Akerholt, G., Hammond, K., Peyton-Jones, S., Trinder, P.: Processing transactions on GRIP, a parallel graph reducer. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93*, number 694 in LNCS, 1993. Springer.
2. Benoit, A., Robert, Y.: Mapping pipeline skeletons onto heterogeneous platforms. In Y. Shi, D. van Albada, J. Dongarra, and P. Sloot, editors, ICCS 2007, number 4487 in LNCS, pages 591–598. Springer-Verlag, 2007.
3. Bisseling, R. H.: *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.
4. Bonorden, O., Juurlink, B., Von Otte, I., Rieping, O.: The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.
5. Chailloux, E., Foisy, C.: A Portable Implementation for Objective Caml Flight. *Parallel Processing Letters*, 13(3):425–436, 2003.
6. Conchon, S., Le Fessant, F.: Jocaml: Mobile agents for Objective-Caml. In ASA'99, pages pages 22–29. IEEE Press, 1999.
7. Di Cosmo, R., Li, Z., Pelagatti, S., Weis, P.: Skeletal Parallel Programming with OcamlP3L 2.0. *Parallel Processing Letters*, 2008.
8. Di Cosmo, R., Pelagatti, S., Li, Z.: A calculus for parallel computations over multidimensional dense arrays . *Computer Language Structures and Systems*, 2005.
9. Gava, F.: External Memory in Bulk Synchronous Parallel ML. *Scalable Computing: Practice and Experience*, 6(4):43–70, 2005.
10. Hammond, K., Trinder, P.: Comparing parallel functional languages: Programming and performance. *Higher-order and Symbolic Computation*, 15(3), 2003.
11. Hayashi, Y., Cole, M.: Bsp-based cost analysis of skeletal programs. In G. Michaelson, P. Trinder, and H.-W. Loidl, editors, *Trends in Functional Programming*, chapter 2, pages 20–28. Intellect, 2000.
12. Hill, J.M.D., McColl, W.F.: BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.
13. Hinsen, K.: Parallel scripting with Python. *Computing in Science & Engineering*, 9(6), 2007.
14. Krusche, P.: Experimental Evaluation of BSP Programming Libraries. *Parallel Processing Letters*, 2008. to appear.
15. Loulergue, F., Gava, F., Billiet, D.: Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In V. S. Sunderam and al., editors, ICCS 2005, volume 3515 of *LNCS*, pages 1046–1054. Springer-Verlag, 2005.
16. Scaife, N., Michaelson, G., Horiguchi, S.: Empirical Parallel Performance Prediction From Semantics-Based Profiling. *Scalable Computing: Practice and Experience*, 7(3), 2006.
17. Skillicorn, D. B., Hill, J. M. D., McColl, W. F.: Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
18. Verlaguet, J., Chailloux, E.: HirondML: Fair Threads Migrations for Objective Caml. *Parallel Processing Letters*, 2008. to appear.