Parallel Superposition for Bulk Synchronous Parallel ML

Frédéric Loulergue

Laboratory of Algorithms, Complexity and Logic
University Paris XII, Val-de-Marne
61, avenue du général de Gaulle – 94010 Créteil cedex – France
Tel: +33 1 45 17 16 50 – Fax: +33 1 45 17 66 01
loulergue@univ-paris12.fr

Abstract. The `BSMLlib` is a library for Bulk Synchronous Parallel programming with the functional language Objective Caml. It is based on an extension of the λ-calculus by parallel operations on a parallel data structure named parallel vector, which is given by intention.
Those operations are *flat* and allow BSP programming in direct mode but it is impossible to express directly divide-and-conquer algorithms. This paper presents a new construction for the `BSMLlib` library which can express divide-and-conquer algorithms. It is called parallel superposition because it can be seen as the parallel composition of two BSP threads which can each use all the processors. An associated cost model derived from the BSP cost model is also given.

Keywords: Bulk Synchronous Parallelism, Functional Programming, Divide-and-Conquer, Cost model

1 Introduction

Some problems require performance carried out by only massively parallel computers of which programming is still difficult. Works on functional programming and parallelism can be divided in two categories: explicit parallel extensions of functional languages — where languages are either non-deterministic [24] or non-functional [11] — and parallel implementations with functional semantics [1] — where resulting languages do not express parallel algorithms directly and do not allow the prediction of execution times. Algorithmic skeleton languages [7, 9, 25, 6, 26, 8], in which only a finite set of operations (the skeletons) are parallel, constitute an intermediate approach. Their functional semantics is explicit but their parallel operational semantics is implicit. The set of algorithmic skeletons has to be as complete as possible but it is often dependent on the domain of application.

Among researchers interested in declarative parallel programming, there is a growing interest in execution cost models taking into account global hardware parameters like the number of processors and bandwidth. The Bulk Synchronous Parallel (BSP) [31, 22, 28] execution and cost model offers such possibilities and with similar motivations we have designed BSP extensions of the λ-calculus [20]

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2659, pp. 223–232, 2003.

and a library for the Objective Caml language, called `BSMLlib` [19], implementing those extensions.

A BSP algorithm is said to be in *direct mode* [12] when its physical process structure is made explicit. Such algorithms offer predictable and scalable performance and BSML expresses them with a small set of primitives taken from the *confluent* BSλ calculus [20]: a constructor of parallel vectors, asynchronous parallel function application, synchronous global communications and a synchronous global conditional.

Those operations are *flat*: it is impossible to express directly parallel divide-and-conquer algorithms. Nevertheless many algorithms are expressed as parallel divide-and-conquer algorithms [29] and it is difficult to transform them into flat algorithms. In a previous work, we proposed an operation called parallel composition [18], but it was limited to the composition of two terms whose evaluations require the same number of BSP super-steps.

In this paper we present a new operation called parallel superposition and its associated cost model. It can be used to write divide-and-conquer algorithms. The presentation of those novelties needs the presentation of the flat `BSMLlib` library and its associated cost model.

We first present the BSP model (section 2). Section 3 is about the "flat" `BSMLlib` library and its cost model. Section 4 introduces a new operation called parallel superposition to the `BSMLlib` library and the cost model associated with this extended library. We then discuss related work (section 5) and conclude (section 6).

2 Bulk Synchronous Parallelism

Bulk-Synchronous Parallelism (BSP) is a parallel programming model introduced by Valiant [31, 28, 22] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP computer contains a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. Its performance is characterized by 3 parameters expressed as multiples of the local processing speed: the number of processor-memory pairs p, the time l required for a global synchronization and the time g for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation in time gh for any arity h.

A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjoint phases. In the first phase each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes. In the second phase the network delivers the requested data transfers and in the third phase a global synchronization barrier occurs, making the transferred data available for the next super-step. The execution time of a super-step s is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

$\text{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} * g + l$ where $w_i^{(s)}$ = local processing time on processor i during super-step s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during super-step s. The execution time $\sum_s \text{Time}(s)$ of a BSP program composed of S super-steps is therefore a sum of 3 terms:$W + H * g + S * l$ where $W = \sum_s \max_i w_i^{(s)}$ and $H = \sum_s \max_i h_i^{(s)}$. In general W, H and S are functions of p and of the size of data n, or of more complex parameters like data skew and histogram sizes. To minimize execution time the BSP algorithm design must jointly minimize the number S of super-steps and the total volume h (resp. W) and imbalance $h^{(s)}$ (resp. $W^{(s)}$) of communication (resp. local computation).

3 "Flat" Functional Bulk Synchronous Parallel Programming

3.1 The `BSMLlib` Library

There is currently no implementation of a full Bulk Synchronous Parallel ML language but rather a partial implementation as a library for Objective Caml. The so-called `BSMLlib` library is based on the following elements.

It gives access to the BSP parameters of the underling architecture. In particular, it offers the function `bsp_p:unit->int` such that the value of `bsp_p()` is p, the static number of processes of the parallel machine. This value does not change during execution.

There is also an abstract polymorphic type `'a par` which represents the type of p-wide parallel vectors of objects of type `'a`, one per process. The nesting of `par` types is prohibited. A type system enforces this restriction.

The BSML parallel constructs operate on parallel vectors. Those parallel vectors are created by:

```
mkpar: (int -> 'a) -> 'a par
```

so that `(mkpar f)` stores `(f i)` on process i for i between 0 and $(p-1)$. We usually write `f` as `fun pid->e` to show that the expression `e` may be different on each processor. This expression `e` is said to be *local*. The expression `(mkpar f)` is a parallel object and it is said to be *global*.

A BSP algorithm is expressed as a combination of asynchronous local computations (first phase of a super-step) and phases of global communication (second phase of a super-step) with global synchronization (third phase of a super-step).

Asynchronous phases are programmed with `mkpar` and with:

```
apply: ('a -> 'b) par -> 'a par -> 'b par
```

`apply (mkpar f) (mkpar e)` stores `(f i) (e i)` on process i. Neither the implementation of `BSMLlib`, nor its semantics prescribe a synchronization barrier between two successive uses of `apply`.

We ignore the distinction between a communication request and its realization at the barrier. The communication and synchronization phases are expressed by:

```
put:(int->'a option) par -> (int->'a option) par
```

where `'a option` is defined by:

```
type 'a option = None | Some of 'a.
```

Consider the expression: `put(mkpar(fun i->fs`$_i$`))` $(*)$

To send a value `v` from process `j` to process `i`, the function `fs`$_j$ at process `j` must be such that (`fs`$_j$ `i`) evaluates to `Some v`. To send no value from process `j` to process `i`, (`fs`$_j$ `i`) must evaluate to `None`.

Expression $(*)$ is evaluated to a parallel vector containing functions `fd`$_i$ of delivered messages. At process `i`, (`fd`$_i$ `j`) evaluates to `None` if process `j` sent no message to process `i` or evaluates to `Some v` if process `j` sent the value `v` to the process `i`.

The full language would also contain a synchronous conditional operation:

```
ifat: (bool par) * int * 'a * 'a -> 'a
```

such that `ifat (v,i,v1,v2)` will evaluate to `v1` or `v2` depending on the value of `v` at process `i`. But Objective Caml is an eager language and this synchronous conditional operation can not be defined as a function. That is why the core `BSMLlib` contains the function: `at:bool par->int-> bool` to be used only in the construction: `if (at vec pid) then... else...` where `(vec:bool par)` and `(pid:int)`. `if at` expresses communication and synchronization phases.

With the above operations, the global control cannot take into account data computed locally. Global conditional is necessary to express algorithms like:

Repeat
  Parallel Iteration
Until Max of local errors $<$ epsilon

This framework is a good tradeoff for parallel programming because: we defined a *confluent calculus* so we designed a purely functional parallel language from it. Without side-effects, programs are easier to prove, and to re-use. An eager language allows good performances ; this calculus is based on BSP operations, so programs are easy to port, their costs can be predicted and are also portable because they are parametrized by the BSP parameters of the target architecture.

3.2 Cost Model

A formal parallel cost model can be associated to reductions in the BSλ-calculus [17]. "Cost expressions" are defined, and each rule of the semantics is associated with a cost rule on cost expressions. Given a particular strategy of reduction, a expression is always reduced in the same way. In this case costs can be associated with expressions rather that reductions.

For the BSλ-calculus it is possible to define two different reduction strategies for the two levels of the calculus. The `BSMLlib` library uses the strategy of the Objective Caml language, the same strategy for local and global reduction: weak call-by-value strategy.

We will not describe the cost of the evaluation of a local expression, it is the same as the cost of the evaluation of the expression by a strict functional language. We give the costs of the evaluation of global expressions.

The cost model associated with our expressions follows the BSP cost model. The evaluation of a parallel vector occurs in the first phase of the BSP super-step. The evaluation of `mkpar` f leads to the evaluation of $(f\ i)$ at each processor i, $0 \leq i < p$. If the sequential evaluation time of each $(f\ i)$ is w_i, the parallel evaluation time of the parallel vector is $max_{0 \leq i < p} w_i$.

Provided the arguments of the parallel application are values, the parallel evaluation time of `apply` $\langle\ f_0\ , \ldots,\ f_{p-1}\ \rangle\ \langle\ v_0\ , \ldots,\ v_{p-1}\ \rangle$ is $max_{0 \leq i < p} w_i$ where w_i is the evaluation time of $f_i\ v_i$ at processor i.

The evaluation of a put operation requires a full super-step. To evaluate `put` $\langle\ f_0\ , \ldots,\ f_{p-1}\ \rangle$, first each processor evaluates the p local expressions $f_i\ j$, $0 \leq j < p$ leading to p^2 values (p per processor) v_{ij} (first phase of the super-step). If the value v_{ij} of processor i is different from `None` it is sent to processor j (communication phase of the super-step). Once all values have been exchanged a synchronization barrier occurs (third and last phase of the super-step) making the values available for the next super-step. At the beginning of this second super-step, each processor i constructs the function (result of the put operation) from the v_{ji} values it has received. Thus provided the argument of the put operation is a value (in this case a parallel vector of values), the parallel evaluation time of `put` $\langle\ f_0\ , \ldots,\ f_{p-1}\ \rangle$ is

$$\max_{0 \leq i < p} w_i^1 \quad + \quad \max_{0 \leq i < p} h_i \times g \quad + \quad l \quad + \max_{0 \leq i < p} w_i^2 \qquad \text{where}$$

- w_i^1 is the evaluation time at processor i of the p expressions $f_i\ j$, $0 \leq j < p$
- $h_i = \max\{h_{i+}, h_{i-}\}$ where h_{i+} (resp. h_{i-}) is the number of words transmitted (resp. received) by processor i. h_{i+} is the sum of the size of the v_{ij} values sent to other processors, and h_{i-} is the sum of the sizes of the values v_{ji} received by processor i from other processors
- w_i^2 is the evaluation time at processor i to construct the result function from the v_{ji} values.

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

The costs (parallel evaluation times) above are context independent. The time required to evaluate a global expression E will be the same in (`fun` $x \rightarrow E'$) E, `put` E, `apply` $E\ E'$, *etc.* This is why our cost model is *compositional.*

4 Parallel Superposition

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

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

The equational semantics of `super` is given by $\texttt{super}\ E_1\ E_2\ =\ (E_1(), E_2())$.

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

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

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

- the cost of the local computations times for each process: $\langle\ w_0\ ,\ldots,\ w_{p-1}\ \rangle$,
- the size of the sent messages : $\langle\ h_0^+\ ,\ldots,\ h_{p-1}^+\ \rangle$
- the size of the received messages : $\langle\ h_0^-\ ,\ldots,\ h_{p-1}^-\ \rangle$

We will note $(\bar{w}, \bar{h}^+, \bar{h}^-)$ the cost of a super-step. The cost of an expression is thus a list of such triples of vectors. If the costs of E_1 and E_1 are respectively:

$$\begin{cases} (\bar{w}^0, \bar{h}^{0+}, \bar{h}^{0-}), \ldots, (\bar{w}^{k_1}, \bar{h}^{k_1+}, \bar{h}^{k_1-}) \\ (\bar{w}'^0, \bar{h}'^{0+}, \bar{h}'^{0-}), \ldots, (\bar{w}'^{k_2}, \bar{h}'^{k_2+}, \bar{h}'^{k_2-}) \end{cases}$$

then the cost of `super` $E_1\ E_2$ is

$$\begin{pmatrix} \langle\ w_0^0 + {w'}_0^0\ ,\ldots,\ w_{p-1}^0 + {w'}_{p-1}^0\ \rangle \\ \langle\ h_0^{0+} + {h'}_0^{0+}\ ,\ldots,\ h_{p-1}^{0+} + {h'}_{p-1}^{0+}\ \rangle \\ \langle\ h_0^{0-} + {h'}_0^{0-}\ ,\ldots,\ h_{p-1}^{0-} + {h'}_{p-1}^{0-}\ \rangle \end{pmatrix}, \ldots, \begin{pmatrix} \langle\ w_0^k + {w'}_0^k\ ,\ldots,\ w_{p-1}^k + {w'}_{p-1}^k\ \rangle \\ \langle\ h_0^{k+} + {h'}_0^{k+}\ ,\ldots,\ h_{p-1}^{k+} + {h'}_{p-1}^{k+}\ \rangle \\ \langle\ h_0^{k-} + {h'}_0^{k-}\ ,\ldots,\ h_{p-1}^{k-} + {h'}_{p-1}^{k-}\ \rangle \end{pmatrix}$$

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

The usual BSP cost of `super` $E_1\ E_2$ is then

$$(\sum_{n=0}^{k} \max\{W^n\}) \ +\ (\sum_{n=0}^{k} h^n) \times g \ +\ k \times l$$

where $\begin{cases} W^n = \max_{0 \leq i < p}\{w_i^n + w_i'^n\} \\ h^n = \max_{0 \leq i < p}\{(h_i^{n+} + h_i'^{n+}) \ , \ (h_i^{n-} + h_i'^{n-})\} \end{cases}$

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

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

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

4.1 Examples

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

```
let scan op vec =
 let rec scan' fst lst op vec =
  if fst>=lst then vec else
  let mid = (fst+lst)/2 in
  let vec'= mix mid (super (fun()->scan' fst mid op vec)
                           (fun()->scan'(mid+1) lst op vec)) in
  let msg vec =
   apply (mkpar(fun i v->
      if i=mid
      then fun dst->if inbounds (mid+1) lst dst then Some v else None
      else fun dst-> None)) vec
  and parop = parfun2(fun x y->match x with None->y|Some v->op v y) in
  parop (apply(put(msg vec'))(mkpar(fun i->mid))) vec' in
 scan' 0 (bsp_p()-1) op vec
```

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

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

```
let parfun f v = apply (replicate f) v
let parfun2 f v1 v2 = apply (parfun f v1) v2
```

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

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

let inbounds first last n = (n>=first)&&(n<=last)
```

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

5 Related Work

There are other libraries based on the BSλ framework. They are based either on the functional language Haskell [23] or on the object-oriented language Python [16]. They propose flat operations similar to ours but no parallel composition.

[30] presents a way to divide-and-conquer in the framework of an object-oriented language. There is no formal semantics and no implementation from now on. The proposed operation is close to our parallel *superposition* (several BSP threads use the whole network) but the programmer has less control over the use of those super-threads. The same author advocates in [21] a new extension of the BSP model in order to ease the programming of divide-and-conquer BSP algorithms. It adds another level to the BSP model with new parameters to describe the parallel machine.

[32] is an algorithmic skeletons language based on the BSP model and offers divide-and-conquer skeletons. Nevertheless, the cost model is not really the BSP model but the D-BSP model [10] which allows subset synchronization. We follow [13] to reject such a possibility. Another algorithmic skeletons language based on the BSP model [14] does not offer divide-and-conquer skeletons. [27] presents another model which allows subset synchronization.

In the BSPlib library [15] subset synchronization is not allowed as explained in [28]. The PUB library [4] is another implementation of the BSPlib standard proposal. It offers additional features with respect to the standard which follows the BSP* model [2] and the D-BSP model [10]. Minimum spanning trees nested BSP algorithms [5] have been implemented using these features.

We also previously worked on a parallel composition [18]. This operation cannot be added to the BSλ-calculus (the obtained systems is no longer confluent) because it is strategy dependent. Parallel superposition is thus the only one to propose a parallel composition which follows the simplest BSP model, which is compositional and which can be added to the BSλ_p-calculus.

6 Conclusions and Future Work

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

Compared to a previous attempt [18], this new construction has not the drawbacks of its predecessor : the two sides of parallel superposition may not have the same number of synchronization barriers ; the cost model is a compositional one ; its semantics is not strategy dependent, it can be added to the BSλ-calculus.

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

Acknowledgments This work is supported by the ACI Grid program from the French Ministry of Research, under the project CARAML (www.caraml.org). The author wish to thank the anonymous referees for their comments.

References

1. G. Akerholt, K. Hammond, S. Peyton-Jones, and P. Trinder. Processing transactions on GRIP, a parallel graph reducer. In Bode et al. [3].
2. W. Bäumker, A. adn Dittrich and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In 3^{rd} *European Symposium on Algorithms (ESA)*, pages 17–30, 1995.
3. A. Bode, M. Reeve, and G. Wolf, editors. *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, Munich, June 1993. Springer.
4. O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library - Design, Implementation and Performance. In *Proc. of 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, San-Juan, Puerto-Rico, April 1999.
5. O. Bonorden, F. Meyer auf der Heide, and R. Wanka. Composition of Efficient Nested BSP Algorithms: Minimum Spanning Tree Computation as an Instructive Example. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2002.
6. G.-H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196:71–107, 1998.
7. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
8. M. Danelutto and D. Ratti. Skeletons in MPI. In S.G. Aki and T. Gonzalez, editors, *International Conference on Parallel and Distributed Computing and Systems*, Cambridge, USA, November 2002. ACTA Press.
9. J. Darlington, A. J. Field, P. G. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In Bode et al. [3].
10. P. de la Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, number 1123–1124 in Lecture Notes in Computer Science, Lyon, August 1996. LIP-ENSL, Springer.

11. C. Foisy and E. Chailloux. Caml Flight: a portable SPMD extension of ML for distributed memory multiprocessors. In A. W. Böhm and J. T. Feo, editors, *Workshop on High Performance Functionnal Computing*, Denver, Colorado, April 1995. Lawrence Livermore National Laboratory, USA.
12. A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.
13. G. Hains. Subset synchronization in BSP computing. In H.R.Arabnia, editor, *PDPTA'98 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 242–246, Las Vegas, July 1998. CSREA Press.
14. Y. Hayashi. *Shaped-based Cost Analysis of Skeletal Parallel Programs*. PhD thesis, University of Edinburgh, 2001.
15. J.M.D. Hill, W.F. McColl, and al. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.
16. K. Hinsen. Parallel Programming with BSP in Python. Technical report, Centre de Biophysique Moléculaire, 2000.
17. F. Loulergue. BSλ_p: Functional BSP Programs on Enumerated Vectors. In J. Kazuki, editor, *International Symposium on High Performance Computing*, number 1940 in Lecture Notes in Computer Science, pages 355–363. Springer, October 2000.
18. F. Loulergue. Parallel Composition and Bulk Synchronous Parallel Functional Programming. In S. Gilmore, editor, *Trends in Functional Programming, Volume 2*, pages 77–88. Intellect Books, 2001.
19. F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In 14^{th} *IASTED International Conference on Parallel and Distributed Computing Systems*, pages 452–457. ACTA Press, 2002.
20. F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.
21. J. Martin and A. Tiskin. BSP Algorithms Design for Hierarchical Supercomputers. *submitted for publication*, 2002.
22. W. F. McColl. Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, 12:265–272, 1996.
23. Q. Miller. BSP in a Lazy Functional Context. In *Trends in Functional Programming*, volume 3. Intellect Books, may 2002.
24. P. Panangaden and J. Reppy. The essence of concurrent ML. In F. Nielson, editor, *ML with Concurrency*, Monographs in Computer Science. Springer, 1996.
25. S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.
26. J. Serot and D. Ginhac. Skeletons for parallel image processing: an overview of the skipper project. *Parallel Computing*, 28(12):1685–1708, 2002.
27. D. B. Skillicorn. Multiprogramming BSP programs. Department of Computing and Information Science, Queen's University, Kingston, Canada, October 1996.
28. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3), 1997.
29. A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, Oxford University Computing Laboratory, 1998.
30. A. Tiskin. A New Way to Divide and Conquer. *Parallel Processing Letters*, (4), 2001.
31. Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.
32. A. Zavanella. *Skeletons and BSP : Performance Portability for Parallel Programming*. PhD thesis, Universita degli studi di Pisa, 1999.