# Bulk Synchronous Parallel ML with Exceptions

Louis Gesbert [a] Frédéric Gava [a] Frédéric Loulergue [b]
Frédéric Dabrowski [c]

[a] *Laboratory of Algorithms, Complexity and Logic, University Paris XII, France*

[b] *Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans, France*

[c] *Institut de Recherche en Informatique et Automatique, Sophia-Antipolis, France*

**Abstract**

Bulk Synchronous Parallel ML is a high-level language for programming parallel algorithms. Built upon OCaml and using the BSP model, it provides a safe setting for their implementation, avoiding concurrency related problems (deadlocks, indeterminism). Only a limited set of the features of OCaml can be used in BSML to respect its properties of safety: this paper describes a way to add exception handling to this set by extending and adapting OCaml's exceptions. The behaviour of these new exceptions and the syntactic constructs to handle them, together with their implementation, are described in detail, and results over an example are given.

*Key words:* Parallel programming, exception handling, functional programming, Bulk Synchronous Parallelism, syntax of languages

## 1 Introduction

The Bulk Synchronous Parallel ML (BSML) language [14] is a parallel extension of ML (a family of functional programming languages). BSML aims at providing the right balance between the two opposite approaches to parallel programming, low-level and subject to concurrency issues, and high-level with loss of flexibility and efficiency. In the former, we find libraries such as MPI [19] generally used with Fortran or C; these approaches are unsafe and leave the programmer responsible for deadlock or indeterminism issues. In the latter stand traditional algorithmic skeletons [3] where programs are safe but limited to a restricted set of algorithms.

BSML follows the BSP (Bulk Synchronous Parallel [16,10,1]) 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 parallel algorithms [11,17,20] and that can be applied to BSML programs.

Exception handling is a traditional and natural mechanism to manage errors and events that disrupt the normal flow of instructions of a program. It can also be used purposefully to extract the results in the course of some recursive algorithms. Widely used languages or libraries for data-parallel programming are mostly imperative like C or Fortran [2,9]. These languages do not provide exception mechanisms. In the case of Java [7], the interaction of parallel constructions with exceptions is not studied. Exception handling is accordingly an issue in parallel languages and efficient, simple and expressive solutions to this problem are a current research topic [18]. To our knowledge, there exists no related work on exception mechanisms for data-parallel languages. The works on exceptions in a concurrent or distributed setting [15] or non data-parallel functional programming languages [8] are not really related to our work which is based on the well-structured parallelism of the BSP model.

BSML is implemented as a library for Objective Caml [12], which enables it to benefit from the advanced, general-purpose features of this language. However, not all of them are compatible with parallelism. Exceptions are one of the features that don't provide the desired safety when used in parallel. In this paper, we adapt and extend the exception handling mechanism of OCaml to respect the constraints of parallel programming in BSML. The approach we define is not specific to OCaml though, and it could be applied to any strict language with exceptions. In particular, Java behaves very similarly to OCaml regarding exceptions and we think there would be little work involved in adapting our system to this language.

In section 2, we introduce the BSP model and Bulk Synchronous Parallel ML (BSML). In section 3 we study issues related to OCaml-style exception handling in a parallel setting, and our solution is presented in section 4. The implementation of this solution for BSML is described in section 5, followed by an example of use and results in section 6. We conclude and introduce future work in section 7.

## 2 Functional Bulk Synchronous Parallel Programming

### *2.1 The BSP Model*

In the BSP model, a computer is a set of uniform *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 (for the sake of conciseness, we refer to [16] for more details). A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjoint phases: (a) Each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes; (b) the network delivers the requested data transfers; (c) a global synchronization barrier occurs, making the transferred data available for the next super-step.

The performance of the machine is characterised by 3 parameters: $p$ is the number of processor-memory pairs, $L$ is the time required for a global synchronization and $g$ denotes the speed of the network. Using these and the structure of the execution, it is possible to predict the performance of a program.

### *2.2 The BSML Language*

| | |
|---|---|
| **bsp_p**: int | **mkpar**: (int $\rightarrow \alpha$ ) $\rightarrow \alpha$ **par** |
| **bsp_g**: float | **apply**: ($\alpha \rightarrow \beta$ ) **par** $\rightarrow \alpha$ **par** $\rightarrow \beta$ **par** |
| **bsp_l**: float | **put**: (int$\rightarrow\alpha$ ) **par** $\rightarrow$ (int$\rightarrow\alpha$ ) **par** |
| | **proj**: $\alpha$ **par** $\rightarrow$ (int $\rightarrow \alpha$ ) |

Figure 1. Primitives

The BSML language is based on seven primitives, three of which are used to access the physical parameters of the machine. A BSML program is built as a sequential program on a parallel data structure called parallel vector. Its type is $\alpha$ **par**, which expresses that it contains a value of type $\alpha$ at each of the $p$ processors, where type $\alpha$ may be any type not containing an occurrence of **par** (this point is discussed in detail in [5]). We adopt the notation $\langle x_0, \ldots, x_{p-1} \rangle$ to denote the parallel vector with value $x_i$ at processor $i$.

BSML programs use the four parallel primitives **mkpar**, **apply**, **put** and **proj** for the creation and manipulation of parallel vectors. The asynchronous computation phase is programmed using the two primitives **mkpar** and **apply**.

**mkpar** creates a parallel vector from a sequential function.

$$\textbf{mkpar}:\ f \mapsto \langle f\ 0, \ldots, f\ (p-1)\rangle$$

This primitive induces *local* computation that may be resolved differently on each processor. We call in comparison *replicated* top-level sequential execution, which is in fact replicated at every one of the processors, and *parallel* execution that involves different values at different processors (*e.g.* parallel vectors and primitives).

The primitive **apply** applies a parallel vector of functions to a parallel vector of arguments:

$$\textbf{apply}:\ \begin{matrix}\langle f_0,\ \ldots,\ f_{p-1}\rangle \\ \langle x_0,\ \ldots,\ x_{p-1}\rangle\end{matrix} \mapsto \langle f_0\ x_0, \ldots, f_{p-1}\ x_{p-1}\rangle$$

The two primitives **put** and **proj** are used for communication, but they also implicitely apply a synchronisation barrier in order to make the exchanged data readily available to the programmer, thus ending the current super-step. This differs from other approaches like BSPlib [9] or PUB [2] where barriers are explicit.

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 (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).

$$\textbf{put}:\ \langle f_0, \ldots, f_{p-1}\rangle \mapsto \left\langle \begin{matrix} f_0\ 0 & & f_0\ (p-1) \\ \vdots & \ldots, & \vdots \\ f_{p-1}\ 0 & & f_{p-1}\ (p-1)\end{matrix} \right\rangle$$

The second communication primitive, **proj**, allows to get replicated values back from locally computed ones. It projects a parallel vector to a standard, replicated vector (again in the form of a function).

$$\textbf{proj}:\ \langle x_0, \ldots, x_{p-1}\rangle \mapsto x_0\ \cdots\ x_{p-1}$$

## 3 Exceptions and BSML

Exceptional situations and errors are handled in OCaml with a powerful system of exceptions. There are two major reasons to use exceptions: first, as a way to quickly get out of a computation and return some parameters. This is specially useful when doing an in-depth search for example, as it saves the trouble of returning the results manually at every level while climbing back in the stack. In parallel, this is at least as relevant since you get the trouble of gathering the results from the different processors. The second reason is error recovery: an unexpected error in OCaml raises an exception. If one processor triggers a Stack_overflow exception during the course of a parallel computation, BSML has to deal with it, like OCaml would, and prevent a crash. This section describes how OCaml handles exceptions and what could get wrong if OCaml exceptions are used in BSML without special care.

**exception** Exc **of** $\tau$ declares a new OCaml exception Exc that encloses data of type $\tau$. Exceptions are considered an extensible variant type, e.g. for matters of pattern-matching. The above-defined exception would be triggered with the syntax **raise** (Exc x), where x is of type $\tau$. Once an exception is raised, it is propagated up the stack until it meets an enclosing **try**...**with** Exc x $\rightarrow$ t block that pattern-matches against the exception. The exceptional behaviour t is then followed and returns a value of the type expected for the expression without exception.

When using this scheme in parallel with BSML, we face three different cases:

(1) If, during a parallel computation, a single processor raises an exception but catches it before the end of the local section, no global operations or communications are hindered and the function that catches the exception returns a result as expected.
(2) Exceptions may be raised during a replicated section. In that case, all processors follow the same path of execution and catch the exception or fail together: no inconsistency appears either.
(3) When an exception is raised locally, but not caught immediately, however, the processor concerned is not going to execute any of the replicated code that might occur until the end of the superstep: the system gets into an inconsistent state. Worse, the concerned processor is most likely not to meet the expected synchronisation at the end of the superstep and cause a deadlock when the other processors reach the barrier.

Let's take a closer look at the last case with an example:

```
let f pid = if pid=0 then raise (Failure "0") else (fun _ → pid)
in let v = mkpar f
in put v
```

| Evaluation at processor 0 | Evaluation at processor 1 |
|---|---|
| **let** v = <**raise** (Failure "0"),..><br>**in put** v | **let** v = <..,**fun** _ → 1,..><br>**in put** v |
| *** Exception raised *** | **put**: trying to send "1" to 0 |

Here, an exception is raised locally on processor 0 but processor 1 continues to follow the main execution stream, until it is stopped by the need for a synchronisation. Then, a deadlock occurs. If the same code had been enclosed in a **try**...**with** Failure →..., processor 0 and 1 would have branched into different global execution streams, the normal one and the exceptional one, leading to a global inconsistency: they could have a different number of super-steps which is not possible in the BSP model.

The solution we provide intends to stay as familiar as possible to the programmer. We explain in the next parts how to extend it to manage the problematic case.

## 4 An Exception Mechanism for BSML

### *4.1 New syntax constructs for exceptions*

The missing piece to a parallel exception system is a way to catch globally exceptions that are raised locally. Exceptions are defined and raised in the usual way from the user side, using the keywords **exception** and **raise**. Only the catching of local exceptions in a replicated setting is changed. Below is an example of use of exceptions in BSML.

```
trypar
  let f pid = match pid with
    | 0 → raise (Failure "0")
    | x → x
  in mkpar f
withpar
  eset → Exception_set.iter
    (fun e → prerr_endline
      (Printexc.to_string e.exc))
    eset
```

The parallel execution of f in this example raises a local exception on processor 0 only. The structure **trypar**...**withpar**, which is similar to **try**...**with** in OCaml is then used to safely recover this local exception, globally. Globalised local exceptions caught this way are implemented as sets (of type Exception_set.t) of records containing the standard OCaml exceptions raised and their originating processor number. Here, **withpar** binds the name eset to a set containing the

Failure raised by processor 0. The exceptional code provided after the arrow iterates on this set and prints the exception on standard error.

The new structure **trypar**...**withpar**, somehow similar to the standard one, is needed mainly for two reasons: first, a formerly-local exception and a standard replicated exception may exist at the same time and need to be distinguished. Second, it deals with *sets* of exceptions and not with single exceptions.

### *4.2 Parallel exception handling mechanism*

We will consider this two points carefully: (a) a local exception should never prevent replicated code from being executed, or the system becomes inconsistent (replicated code is not executed by all the processors anymore); (b) at the end of the super-step, a local exception has to be treated replicatedly.

Since replicated and local code may be juxtaposed in the same super-step, we need to get aside from the standard exception handling techniques to ensure that replicated code is run normally even after a local exception. During a super-step, there might be local and replicated exceptions coexisting and they must be treated at different levels: a replicated exception, since it is raised by all processors, is treated immediately in the OCaml way. A local exception, on the other hand, must not hinder the global behaviour of the processor yet, so it is kept silent to replicated code until the end of the super-step. This means, in particular, that a processor in a state of exception may not perform any local computation until the next synchronisation.

At the end of a super-step (**put** or **proj**), the exception state is communicated to all processors to allow a global decision to be taken. In such a situation, the local results obtained are partial, inconsistent or nonexistent. Although we are discussing a way to enable the program to recover them afterwards, we currently adopt the standard approach and discard them, switching to the exceptional treatment specified by the user.

Local exceptions are thus *deferred* until the end of the super-step. However, it is undesirable that an exception escapes the scope of the **trypar**...**withpar** it was raised in. For this reason, communications (and barrier) must be forced at the **withpar**.

This behaviour is described formally and in more detail in the semantics presented in [6].

## 5 Implementation

Keeping local exceptions hidden from replicated execution is made possible by the strict distinction between local and replicated execution in BSML; the implementation relies on standard OCaml exception handling to deal with local exceptions. Any local execution is enclosed **try**...**with** safety net in the implementation of the parallel primitives, ensuring that we catch any exception that is raised on a single processor. These exceptions can't be ignored in further local computation on that processor though, so they are retained in a local variable :

status: (Fine | Stopped **of** int ∗ exn) **ref**.

The value of this reference indicates what state the processor is in; in the Stopped state, all primitives that perform local computation will simply ignore them, as they could try to use results that didn't compute. Replicated execution, on the other hand, is not affected.

At the end of the super-step, initiated by the **put** or **proj** primitives (or by **withpar**), the communication phase starts with an exchange of data sizes. We take the opportunity to communicate processor states: in case there is any exception, normal communication is replaced by a total exchange of the exceptions and their parameters. Then, a new super-step starts and the processor states are reset to fine. This is consistent w.r.t. replication, because the same set of exception is raised everywhere.

The set of local exceptions has yet to be raised. As propagation of exceptions drives out of the normal functional execution flow and can't be implemented in OCaml (without exceptions) without modifying the internals of the compiler, the set of exceptions is piggy-backed onto an OCaml exception Global_exn **of** Exception_set.t.

To implement the extension in the language, in particular the new keywords **trypar** and **withpar**, we chose to use OCaml's generic precompiler, camlp4. The **trypar**...**withpar** structure is replaced by some code including a **try**...**with** over exceptions of the kind Global_exn. Several critical points make the rewriting rules actually more complicated than that:

- **withpar** should do a barrier to gather local exceptions before it can catch them with **with** Global_exn eset →....
- The user can catch any exceptions, including Global_exn, with the default pattern _. A rewriting rule adds a pattern-matching rule that re-raises Global_exn before the default pattern when this is the case.
- There may be nested **trypar**...**withpar** structures. We use de Bruijn indices to prevent an inner **withpar** from catching local exceptions raised out of its

scope.
- Local exceptions and global exceptions may conflict. A native global exception is filtered by **withpar**, that still performs a barrier, to ensure it was not raised after a local exception that should take precedence.

To summarize, the implementation is a combination of an added protection on the primitives, an extended communication protocol that can be triggered to global exchange of exceptions, and various rewriting rules that both add new constructs and ensure the non-interference of this mechanism with the user program.

## 6 Experiments

### *6.1 A generic parallel backtracking algorithm*

Backtracking consists in searching for a solution by exploring a tree of possibilities depth-first. If a recursive function doing this search raises an exception whenever a solution is found, it can be caught directly by the calling function without the need to switch cases and return a solution if it exists, or continue exploring otherwise. The parallelisation of this process explores the children of several different nodes at the same time, making the gathering of solutions even more difficult without using exceptions.

To assess the usability of exceptions in BSML, we present a very simple implementation of generic parallel backtracking. It takes as argument a sequential function f_chld that returns all the children of a given node in the tree and raises a specific exception on a solution. The backtracking function is shown below in pseudo-OCaml code. We suppose that scatter and gather split a list into a parallel vector and back, respectively.

```
let rec backtrack f_chld nodes =
  let vec = scatter nodes in
    for v = take a limited amount of the nodes in vec at each processor, do
      let chld_v = locally, map f_chld to v
      in backtrack f_chld (gather chld_v)
    done
```

This recursive function is initially called on f_chld of the root of the tree, within a **trypar**. . . **withpar** that gathers and returns any solution found.

Another version, without any use of exceptions, was implemented. The main

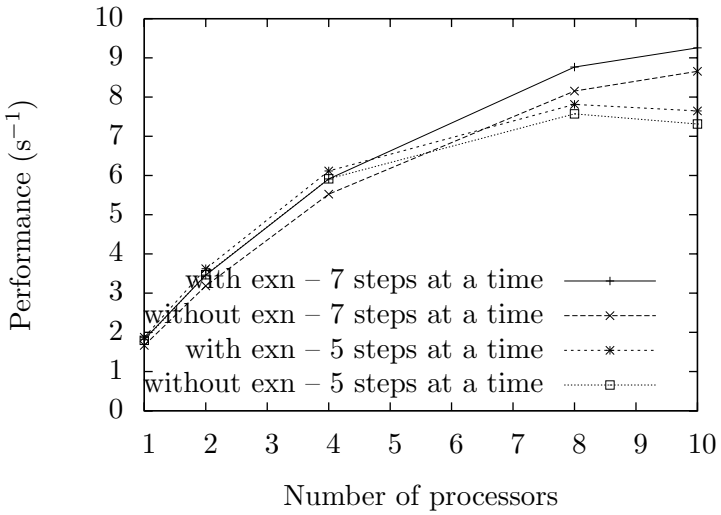


Figure 2. $9 \times 9$ sudoku solved with and without exceptions

descending function had to gather the results of all processors and check if there was a solution at one of them. Accordingly, the size of the backtrack function was increased from 26 lines of code to 44 – exceptions made us save 40% in code size on this example.

### *6.2 Results*

As an example, we used this backtracking algorithm to solve sudoku with brute-force. Sudoku is a fashionable game that consists in filling a $n^2 \times n^2$ grid with integers from 1 to $n^2$ according to constraints that ensure, given some initial numbers, that only one solution is possible. The children are generated by trying every number at each free square and checking for validity. A mild optimisation consists in composing the children function several times to obtain enough nodes for an even distribution between processors, which becomes mandatory when increasing the size of the machine.

We solved a given grid of dimensions $9 \times 9$ using native-code execution on a cluster of PC linked with a gigabit network, for a number of processors varying from one to ten. Figure 2 shows the performance in seconds$^{-1}$ depending on the number of processors (so that a linear speedup would be a straight line), for two different levels of the latter optimisation. This results are the median of a large number of experiments. We notice little impact on performance between the versions with and without exceptions, which is sound since the algorithm is not changed; better, the difference is very stable and in favor of the version with exceptions: we explain it by the added checks that have to be made to extract the possible results at every step of computation.

## 7 Conclusion and Future Work

Parallel architectures are taking the lead in computer hardware. Advanced programming paradigms, however, are still trying to find the best expression for the adapted programs. In this paper, we tackled the problem of exception handling for the functional, OCaml-based BSML language, pushing it one step further to that goal.

We defined global sets of locally raised exceptions and dedicated handlers which offer a natural way to deal with them. A realistic implementation was presented, together with a test program and promising benchmarks.

The work presented here is tightly related to the BSP model, but the exception scheme it is based on is not specific to OCaml. Hence, we reckon there would be little work involved in translating it to, for instance, a Java implementation of BSP. Future work includes recovery of partial results, a full type system which could include detection of uncaught exceptions [4,13], and automated performance prediction.

## References


[1] R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.

[2] O. Bonorden, B. Juurlink, I. von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[4] P. de Groote. A simple calculus of exception handling. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Typed Lambda Calculi and Applications (TLCA)*, volume 902 of *LNCS*, pages 201–215. Springer, 1995.

[5] 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.

[6] L. Gesbert and F. Loulergue. Semantics of an Exception Mechanism for Bulk Synchronous Parallel ML. In *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 201–208. IEEE Computer Society, 2007.

[7] Yan Gu, Bu-Sung Lee, and Wentong Cai. JBSP: A BSP programming library in Java. *Journal of Parallel and Distributed Computing*, 61(8):1126–1142, 2001.

[8] K. Hammond. Exception handling in a parallel functional language: PSML. In *Proceedings of the IEEE TENCON '89, Bombay*, pages 169–173. IEEE, 1989.

[9] J.M.D. Hill and W.F. et al. McColl. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.

[10] Jonathan M. D. Hill and David Skillicorn. Lessons learned from implementing BSP. *Future Generation Computer Systems*, 1998.

[11] Yuguang Huang and W F McColl. A Two-Way BSP Algorithm for Tridiagonal Systems. *Future Generation Computer Systems*, 13:337–447, 1998.

[12] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System release 3.10, 2007. web pages at caml.inria.fr.

[13] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000.

[14] F. Loulergue, F. Gava, and D. Billiet. BSML: Modular Implementation and Performance Prediction. In *Proc. of ICCS*, LNCS 3515, pages 1046–1054. Springer, 2005.

[15] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. *ACM SIGPLAN Notices*, 36(5):274–285, May 2001.

[16] W. F. McColl. Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, 12:265–272, 1996.

[17] R. O. Rogers and D. B. Skillicorn. Using the BSP cost model to optimise parallel neural network training. *Future Generation Computer Systems*, 14(5-6):409–424, 1998.

[18] A. B. Romanovsky, C. Dony, J. Knudsen, and A. Tripathi, editors. *Advances in Exception Handling Techniques*, volume 2022 of *LNCS*. Springer, 2001.

[19] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.

[20] A. Tiskin. Communication-efficient parallel generic pairwise elimination. *Future Generation Computer Systems*, 23(2):179–188, 2007.