

BULK SYNCHRONOUS PARALLEL ML WITH EXCEPTIONS

Louis Gesbert¹, Frédéric Gava¹, Frédéric Loulergue²
and Frédéric Dabrowski³

¹*Laboratory of Algorithms, Complexity and Logic, University Paris XII, France*

²*Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans, France*

³*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.

Keywords: Parallel programming, exception handling, functional programming, BSP, syntax of languages

1. Introduction

The Bulk Synchronous Parallel ML (BSML) language [9] 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 [12] 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 [1, 11]) 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 BSML with a simple and efficient cost model.

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, 7]. These languages do not provide exception mechanisms. In the case of Java [6], 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 [10]. To our knowledge, there exists no related work on exception mechanisms for data-parallel languages.

BSML is implemented as a library for Objective Caml [8], which enables it to benefit from the advanced, general-purpose features of this language. A few of these features however, among which exception handling, do not 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 [11] 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 : unit→int bsp_g : unit→float bsp_l : 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)
---	---

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 α **par**, which expresses that it contains a value of type α at each of the p processors, where type α may be any type not containing an occurrence of **par** (this point is discussed in detail in [4]). We adopt the notation $\langle x_0, \dots, x_{p-1} \rangle$ 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.

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

This primitive induces *local* computation that will 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:

$$\mathbf{apply}: \begin{array}{l} \langle f_0, \dots, f_{p-1} \rangle \\ \langle x_0, \dots, x_{p-1} \rangle \end{array} \mapsto \langle f_0\ x_0, \dots, f_{p-1}\ x_{p-1} \rangle$$

Unlike BSPLib [7] or PUB [2] we do not distinguish between communication phase and synchronization barrier. The two primitives **put** and **proj** both end implicitly with a synchronization barrier, putting an end to the current super-step.

put is the first communication primitive. 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.

$$\mathbf{put}: \langle f_0, \dots, f_{p-1} \rangle \mapsto \left\langle \begin{array}{ccc} f_0 \ 0 & & f_0 \ (p-1) \\ \vdots & & \vdots \\ f_{p-1} \ 0 & \dots, & f_{p-1} \ (p-1) \end{array} \right\rangle$$

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.

$$\mathbf{proj}: \langle x_0, \dots, x_{p-1} \rangle \mapsto x_0 \ \dots \ x_{p-1}$$

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 τ declares a new OCaml exception `Exc` that encloses data of type τ . 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 τ . 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 _ ->Some pid)
in let v = mkpar f
in put v
```

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

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.

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

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 A new 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 [5].

5. Implementation

Keeping local exceptions hidden from replicated execution is made possible by the strict distinction between local and replicated execution in BSML: by enclosing local execution in a **try...with** safety net in the implementation of the parallel primitives, we are sure to catch every exception 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 second role of the safety net is then to prevent any local operation on that processor until the end of the super-step, since these operations may use results that failed to compute; replicated code will continue to execute normally.

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. We then get back to a consistent replicated state with the same set of exceptions raised everywhere. The propagation of exceptions drives out of the normal execution flow and can't be implemented in OCaml (without exceptions) outside of the compiler, it is therefore piggy-backed onto the 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 core of **trypar...withpar** is a **try...with** catching exceptions of the kind `Global_exn`, but several other problems must be taken into account:

- A barrier must be done before the **with**.
- The super-step may end at an imbrication level different from the one the exception was raised at.
- Local exceptions and global exceptions may conflict. A native global exception has to meet the **withpar** barrier before jumping further in the stack.

- Our `Global.exn` must be protected from being caught by the user with a normal `try...with`.

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 simple implementation of generic parallel backtracking. It takes as argument a sequential function that returns all the children of a given node in the tree and raises a specific exception on a solution. The function exploring the tree proceeds in three steps:

- 1 the list of current nodes is split into a parallel vector
- 2 the function returning the children is run in parallel on a limited amount of the nodes at each processor
- 3 the resulting children nodes are gathered globally, and these three steps are processed recursively on them.

If no solution has been found at the third step (no exception raised), new nodes from step 2 are tried. If there are none left, the algorithm backtracks to the caller. This recursive function is enclosed into a `trypar...withpar` that gathers and returns any solution found.

Another version, without any use of exceptions, was implemented. The main 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 core part of the algorithm was increased from 26 lines of code to 44 – exceptions made us save 40% in code size on that example.

6.2 Results

As an example of use, we implemented a brute-force sudoku solver. 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. We generate the children by trying every possibility for each free square and checking for validity. A mild optimisation consists in composing the children function

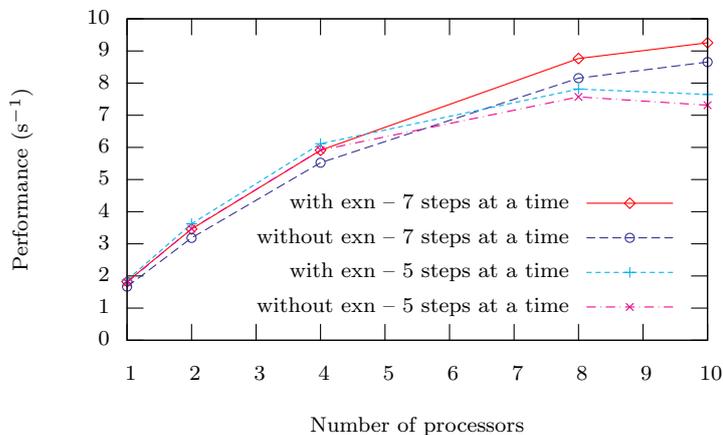


Figure 2. Sudoku of dimension 9 solved with and without exceptions

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 dimension 9 on a cluster of PC (using native-code execution) 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

Hardware is heading massively towards parallel architectures. 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 bases on is not specific to OCaml. Hence, we reckon there would be little work involved in translating it to, for instance, Java. Future work includes recovery of partial results, a full type system, 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] 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.
- [5] L. Gesbert and F. Loulergue. Semantics of bulk synchronous parallel ml with exceptions. In Zoltán Horváth, editor, *Draft proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL'06)*. to appear, 2006.
- [6] 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, August 2001.
- [7] J.M.D. Hill, W.F. McColl, and al. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.
- [8] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System release 3.09, 2005. web pages at www.ocaml.org.
- [9] F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In Vaidy S. Sunderam, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *International Conference on Computational Science, Part II*, number 3515 in LNCS, pages 1046–1054. Springer, 2005.
- [10] Alexander B. Romanovsky, Christophe Dony, Jørgen Lindskov Knudsen, and Anand Tripathi, editors. *Advances in Exception Handling Techniques (the book grew out of a ECOOP 2000 workshop)*, volume 2022 of *Lecture Notes in Computer Science*. Springer, 2001.
- [11] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
- [12] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.