New syntax
of a High-level BSP language
Application to parallel pattern-matching and exception handling

Louis Gesbert & Frédéric Gava

June 2009
TR–LACL–2009–6

Laboratoire d’Algorithmique, Complexité et Logique (LACL)
University of Paris–East

Université Paris 12 – Val de Marne, Faculté des Science et Technologie
61, Avenue du Général de Gaulle,
94010 Créteil cedex, France
Tel.: (33)(1) 45 17 16 47, Fax: (33)(1) 45 17 66 01
New syntax
of a High-level BSP language
Application to parallel pattern-matching and exception handling

Louis Gesbert Frédéric Gava
Laboratory of Algorithmics, Complexity and Logic (LACL)
University of Paris–East (Paris 12),
61 avenue du Général de Gaulle, P2 du CMC
94010 Créteil cedex, France
louis.gesbert@univ-paris-est.fr and frederic.gava@univ-paris-est.fr

Abstract

Bulk-Synchronous Parallel (BSP) ML is a high-level language for programming parallel algorithms. Built upon OCaml, it provides a safe setting for the implementation of BSP algorithms and avoiding concurrency related problems (deadlocks, indeterminism etc.). Currently, BSML is based on a very small core of parallel primitives that extended ML sequential programming to BSP one. But we found that currently the price was to read programs with hardness. We have thus choose to design a new syntax that makes programs easier to read and so to debug. This new syntax also gives us a smart and uniform syntax for parallel patterns and exceptions handlers in BSML. In this paper, we present the problem of the past syntax, the new one, matching of parallel values and exceptions. Implementations are also detailed and examples are given to show the useful of the work (and BSML). In final, some benchmarks complete this article.
# Contents

1 Introduction ................................................................. 1
   1.1 Generalities ......................................................... 1
   1.2 BSML ................................................................ 1
   1.3 A new and more elegant syntax .................................. 2
   1.4 BSML and patterns and exceptions .............................. 3
   1.5 Related works .......................................................... 3
   1.6 Outline ................................................................. 4

2 Past BSP Programming in ML ............................................. 5
   2.1 The Bulk-Synchronous Parallel Model ......................... 5
   2.2 Bulk-Synchronous Parallel ML (BSML) ......................... 5
      2.2.1 General description ........................................... 5
      2.2.2 Model of Execution .......................................... 6
      2.2.3 Parallel primitives ........................................... 7
      2.2.4 Other features .................................................. 8
   2.3 Examples ............................................................... 8

3 New Syntax .................................................................... 13
   3.1 Main problem .......................................................... 13
   3.2 General description .................................................. 13
   3.3 Implementation ....................................................... 14
   3.4 Disadvantages of this new syntax ................................. 15
   3.5 Small examples ........................................................ 16
      3.5.1 Parallel Sampling Sort Algorithm ......................... 16
      3.5.2 Parallel reducing .............................................. 16
      3.5.3 Sieve of Eratosthenes ....................................... 16
      3.5.4 The \( N \)-body problem ..................................... 18
      3.5.5 Dense Matrix Multiplication ............................... 19

4 Pattern matching of parallel vectors ................................. 23
   4.1 General ideas ........................................................... 23
   4.2 Exceptions ............................................................. 24
      4.2.1 Problems .......................................................... 24
      4.2.2 Patterns for exceptions ...................................... 25
   4.3 Implementation ........................................................ 25

5 Full example ................................................................... 29
   5.1 Parallel backtracking ................................................ 29
      5.1.1 Description of the problem .................................. 29
      5.1.2 Sequential algorithm ......................................... 29
      5.1.3 Naive parallel version ........................................ 29
   5.2 Efficient parallel version ............................................ 30
6 Conclusion

6.1 Conclusion ................................................................. 35
6.2 Future work .............................................................. 35
## List of Figures

2.1 A BSP super-step (left) and BSML model of execution (right) ........................................... 6  
2.2 Summary of BSML primitives ............................................................................................... 7  
2.3 The evaluation of the superposition for two super-threads on 3 processors .......................... 8  
2.4 Parallel reduce in past (left) and the same using the new syntax (right) ............................ 11  
2.5 Divide-and-conquer reduce computation (left) and the same using the new syntax (right) ... 11  
2.6 Parallel sampling sorting in past (left) and new (right) syntax ............................................. 11  
2.7 Benchmarking of reducing algorithms (addition of polynomials) ....................................... 12  
2.8 Benchmarking of the PSRS sorting of lists ........................................................................... 12  
3.1 Summary of BSML alternate syntax ....................................................................................... 14  
3.2 3 different Sieves of Eratosthene (left) and the same using the new syntax (right) .............. 17  
3.3 BSML code for the \(N\)-body problem (left) and the same using the new syntax (right) ....... 19  
3.4 BSML code of the matrix multiplication (left) and the same using the new syntax (right) .... 20  
3.5 Generic systolic loop (left) and the same using the new syntax (right) ................................. 21  
3.6 Shifting values of processors (left) and the same using the new syntax (right) ...................... 21  
3.7 Communications in the matrix multiplication (left) and the same using the new syntax (right) 21  
3.8 Performance of the \(N\)-body problem .................................................................................. 21  
3.9 Performances of the sieve of Eratosthenes .......................................................................... 22  
3.10 Execution time of multiplications of dense matrices of complex ........................................... 22  
4.1 Abstract syntax of parallel patterns ....................................................................................... 24  
4.2 Example of the parallel pattern transformation ..................................................................... 26  
5.1 Parallel and generic backtracking methods using parallel patterns ....................................... 31  
5.2 Code of the rebalancing function ......................................................................................... 31  
5.3 Code of the “if needed” rebalancing function ....................................................................... 32  
5.4 Speed of execution for solving a \(16 \times 16\) sudoku (backtracking) ........................................... 33  
5.5 Same speed of execution versus linear acceleration ............................................................... 33
Chapter 1

Introduction

1.1 Generalities

Since the paper “Go To Statement Considered Harmful”, structured sequential programming is the norm. It is surprising to see that it is absolutely not the case for parallel programming [26]. Besides compiler-driven automatic parallelisation or parallel loops (as in the Microsoft’s Task Parallel Library [30]), programmers have kept the habit of using low-level parallel routines (as send/received of MPI/PVM or else) or a concurrent language. In this way, they, less or more, managed the communications with the usual problems of (un)buffer or (un)blocking sending, managed concurrent accesses to share resources etc., which are all source of deadlocks and non-determinism.

Furthermore, programmers forbid optimisations that could be done if more high-level structures as collective operators or skeletons were to be used instead.

In the context of “Think Parallel or Perish” that is an increasing pervasiveness of multi-CPU/cores systems for everywhere, parallel code would be the norm. But many programmers are not able to manipulate these low-level routines without introduce many bugs. Some experts exists but they can not parallel all the codes.

High-level models, languages and tools are thus needed but sadly rarely be used. The main reason of this fact is that they do generally not offer a sufficiently wide set of parallel structures (as skeletons [9]) for a practical and efficient programming.

That makes the design of new and robust parallel programming languages an important aera of research. Creating such a language involves a tradeoff between the possibility to write predictable and efficient programs and the abstraction of such features to make programming safer and easier.

1.2 BSML

An interesting compromise is Bulk-Synchronous Parallel ML (a.k.a. BSML), an extension of ML (a family of high-level programming languages: functional, modular and imperative) to code bulk-synchronous algorithms [5] which combines the high degree of abstraction of ML (without poor performances because many time ML programs are as efficient as C ones) with the scalable and predictable performances of BSP.

BSP is a parallel model which offers a high degree of abstraction and allows an estimation of the execution time of its algorithms on a wide variety of architectures as massively parallel machine (Cray etc.) or bewolf clusters of PCs or multi-core PCs or mix of the two above architecture.

Against all the difficulties introduced by the low level parallel models which return parallelism too difficult for the majority of the programmers, the BSP community have take the choice of a pragmatic approach: using a paternalist model which facilitates the comprehension of the parallel codes and thus limits the errors under-unclaimed with parallelism (deadlock, bottleneck etc). They also design BSP extensions for traditional programming languages as the form of set of parallel routines defined in a library.

Indeed, in this structural model, programs are written as a sequential sequence of steps, each alternates a phase of computation, a phase of communication and a finish with a global barrier. Communications are bulk and

---

1 These properties are justified for concurrent computations but clearly not for parallel algorithms, i.e high-performances applications.
2 We refer to [5] for a gentle introduction to the BSP model.
collective. That simplify parallel programs because programmer is not responsive to managed how sending data are packaged, routed and received by other processors. Also, within a super-step, the work is done in parallel but the global structure of the algorithm is sequential. This simple structure has proven its worth in practice in many parallel applications.

BSML is thus an extension of ML to code BSP algorithms using a small set of high-level primitives. It 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 [37] 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 [9] where programs are safe but limited to a restricted set of algorithms.

With BSML, only 6 primitives are necessary to parallelism (9 if we count functions for accessing to the 4 BSP hardware parameters). 4 will be necessary with the new syntax presented in this article. We thus think, that makes easier parallel programming for ML experts to beginners. By comparison, the standard BSPlib [7, 36] for BSP programming in C needs about fifteen routines, which was already little compared to the hundred of MPI (even if the majority of the programs do not use all these functions [3]).

Furthermore, BSML would explicitly force the programmer to have the global and sequential sequence of super-steps of the BSP paradigm. This is more paternalistic as the original BSPlib library. That ensure a better safety which is very important for the ML community (more than performance). Indeed, BSPlib is SPMD (“single program, multiple data”) programming. As in MPI, the programmer can write:

\[
\text{if pid==0 then sync() else asynchronous_computation().}
\]

i.e., processor 0 begins a global barrier of synchronisation and other processors finish their works. A deadlock would clearly occur. In fact, synchronisations that finish each super-step can be done in very different points of the code and this for each processor. A good programmer do not write code as it because that break the sequential and global structure of the BSP paradigm.

BSML follows this 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.

Using BSML, the programmer is forced to write its program as a global sequence that spawn parallel works. At the syntax level, few entry points are needed for parallelism. BSML is based on a data-type called parallel vector which, among all OCaml types, enables parallelism. A parallel vector embeds \( p \) (number of processors of the BSP machine) values of any type at each of the \( p \) different processors in the parallel machine. Parallel vectors are handled through the use of five different primitives that constitute the core of BSML: two asynchronous ones; two synchronous ones that perform communications; and one to parallel composition of BSP computations. Therefore, BSP also provides a simple and efficient cost model which is particularly helpful in the design of efficient parallel algorithms [12, 5, 27, 4, 35, 13, 8, 29] and that can be applied to BSML programs.

### 1.3 A new and more elegant syntax

BSML is currently implemented as a library for the Objective Caml (a.k.a. OCaml) language, which enables it to benefit from the advanced, general-purpose features of this language.

But BSML programming with only these general primitives is (exaggerating a bit) quickly laborious as lambda-calculus. So, most of time, particularly for communications part of the parallel algorithms, pre-defined communications functions (as total exchange, broadcasts etc.) are preferred to primitives. These functions are part of a standard library for BSML. Communications primitives are thus only used when the pattern is very depending of the algorithm.

---

1. Without counting the advantages in term of performances of BSP programs [36, 5]: sending messages will be in a bulk manner and thus more effective than by small packages; semantics is much clearer which makes it possible to better understand programs and to find the potentials errors: it becomes useless to seek a precise order for the asynchronous send/received so to avoid deadlocks.


3. See [http://ocaml.org](http://ocaml.org) for more details.
Nevertheless, for non-communication parts (we recall that BSML programming looks like data-parallelism) of the programs, primitives are used many times. Some pre-defined functions exist but they look too much the primitives (or do not permit a better read of the programs) and it seems too hard to find better ones. For example, makes computations different following the “pid” of the processor implies each time to call a primitive. And the same problem happens when the programmer wants to used a parallel data in different parts of the code (section 3 presents formally the problems). Some readers of our past papers point out this fact.

Having a small core of parallel operations is a great strength for the formalisation of the language. It makes the definitions clear and allows proof of programs [15]. But we found that the price was to read programs with hardness.

We have thus make the choice to design a new syntax that is just a syntactic sugar to the primitives. Even if syntax is just syntax and do not modify the principle of programming, it allows a simple read of the BSML programs (even for non-ML programmers). For just users, we believe it is as important as semantics. The main idea is to write explicitly the construction of the parallel vector (as lists or arrays can be explicitly written in OCaml) and use a special call in the code to get the value of a parallel vector inside another one.

It have also a nice side-effect: its give us a smart syntax for parallel patterns and exceptions handlers in BSML.

1.4 BSML and patterns and exceptions

Pattern matching is an important feature of high-level programming languages (OCaml, Java etc.) It offers sub-term extraction and function definition by cases. One problem with the matching of parallel values is to define suitable and useful patterns. In our opinion, the new syntax will gives us an interesting way to do that.

Exception handling is a traditional and natural mechanism to manage errors and events that disrupt the normal flow of instructions of a program. It is widely used in high-level languages as Java, Python, ML etc. It can also be used purposefully to extract the results in the course of some recursive algorithms.

In OCaml, syntax to catch exceptions is the same as pattern matching : it is a match of what exceptions could be raised in a sub-part of the code. In the case, of BSML, exceptions need an extra treatments : only a subset of the processors can trigger an exception (for example “Stack_overflow”) during the course of the parallel computation and BSML has to deal with it, like OCaml would, and prevent a crash. This intuitively generate to manage a set of local exceptions (we refer to [23][22] for implementation issues and [24][22] for a formal semantics) which could be thus represented as a parallel vector. In [23][22][24] the set of parallel exception is represented by a variable (of type set) which is hard to manipulate : programmer have to iterate on this set each time he wants to find an appropriate exception. A coherent syntax is thus need for both pattern matching and exception handling.

Note that the approach we define is not specific to OCaml though, and it could be applied to any strict language with BSP primitives and exceptions or patterns. In particular, Java and Python behave very similarly to OCaml regarding exceptions and BSP’s implementation exist for Java [6] and Python [28]. Moreover, the second were design in a BSML spirit by explicitly build an object as working as a parallel vector. We think there would be little work involved in adapting our new syntax and our system to these languages.

1.5 Related works

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.

We are also not aware of a data-parallel language that have a syntax close to our nor for matching parallel values. Many skeletons language as OCamlP3l [10] have a special syntax to compose their skeletons (as pipe, map, farm etc.) but this looks more functional patterns than our programming syntax.

As certain readers could point out it, the notation ≪ e ≫ introduced in this paper and which built a parallel vector, is close to the construction like “e1 par e2” of "parallel Haskell" [40] (if using it p − 1 time) or creation of process as in Eden [31]. But the meaning is not the same one.

Fine-grained parallelism introduced by these expressions take two arguments that are to be evaluated in parallel. The expression “e1 par e2” has the same value as “e2”. Its dynamic behaviour is to indicate that “e1” could
be evaluated by a new parallel thread, with the parent thread continuing evaluation of “e2”. Threads are then distributed on the processors at run-time. Communications are implicit by the share of variables (seen like channels of communication).

In our case, parallelism is explicit (as well as the communications and the distribution of data) and especially it is prohibited to nested parallelism to optimize performances of the implementation [22, 14].

BSML is clearly lower level programming language but this sees allotting with a realistic cost model (BSP) and is more close to the writing of most of coarse-grain algorithms.

Elsewhere, BSML can serve to the implementation for this higher-level kind of parallel language as that was already done for the algorithmic skeletons [20]. This allowing user to choose their programming paradigm: skeletons or "par/seq" are not always very effective or practical for certain tasks [9, 1]. A contrario that is also true for BSML but is still keeping a good prediction of the performances.

First intuitive idea to our parallel vector patterns comes from those of arrays of rewriting languages such as TOM (for Java and C) [2] and MGS [25] (for OCaml). Both use regular expressions as patterns for arrays. For example, 0*+1* match an array containing 0 or 1 everywhere. Our syntax for pattern matching of vectors looks like regular expressions but is more powerful without harder to read due to the fixed size nature of the vectors and to the fact that there are no real order between elements in the vectors: each value in the vector is just a value contain by only one processor. So, they can be seen as sets with a fix cardinal.

1.6 Outline

This paper describes our new syntax and application to pattern matching of parallel values: parallel vectors and sets of parallel exceptions.

First, we briefly present in Chapter 2 the BSML language in its past syntax and some useful functions. In Chapter 3 we describe the new syntax and comparisons of some examples between the past and new syntax. Chapter 4 presents how using this syntax for parallel pattern matching. Chapter 5 is devoted to the implementation of a bigger example with some benchmarks. Future works and conclusion are discussed in Section 6.

Note that all our performance evaluations where done in this paper using the new syntax and MPI. We refer to [19] for some benchmarks using other implementations and comparisons.

A good familiarity with ML programming is assumed, but no prior exposure to functional languages is required. We refer to the manual of OCaml (which also provide books for beginner) for a tutorial introduction to the language.
Chapter 2

Past BSP Programming in ML

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 [5, 36].

A BSP program is executed as a sequence of super-steps (see left scheme in Fig. 2.1), each one divided into three successive disjoint phases: each processor only uses its local data 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 execution time (cost) of a super-step is the sum of the maximal of the local processing, the data delivery and the global synchronisation times.

The performance of the BSP machine is characterised by 4 parameters that could be benchmark [5] to determine execution time of BSP programs:

1. the local processing speed \( r \);
2. the number of processor \( p \);
3. the time \( L \) required for a barrier;
4. and the time \( g \) 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 \( g \times h \). 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. The cost of a program is the total sum of the cost of its super-steps.

2.2 Bulk-Synchronous Parallel ML (BSML)

2.2.1 General description

BSML is currently just a library based on the Objective Caml (OCaml) language; this choice was made among the different variants of ML available mainly for a reason of efficiency, since we target high-performance computation. Other reasons include the amount of libraries available and the tools provided. In particular, we make an extensive use of Camlp4, the Caml Preprocessor and Pretty-Printer to implement our extensions to the language. We plan a full language implementation by generated adequate OCaml code.

The core syntax of BSML is that of OCaml – with few restrictions. BSML programs can mostly be read as OCaml ones, in particular, the execution order should not seem unexpected to a programmer used to OCaml, even though the program is parallel. Moreover, most normal OCaml programs can be considered as BSML programs that do not make use of parallelism: the programs are executed sequentially on each processor of the parallel machine and return their results normally. This is useful when porting programs to BSML and allows the parallelisation to be done incrementally from a sequential program.
At the syntax level, few entry points are needed for parallelism. BSML is based on a datatype called parallel vector which, among all OCaml types, enables parallelism. A parallel vector has type `a par and embeds values of any type `a at each of the different processors in the parallel machine. is defined as a constant throughout the execution of the program. We use the following notation to describe a parallel vector: $\langle x_0, x_1, \ldots, x_{p-1} \rangle$.

### 2.2.2 Model of Execution

What distinguishes this structure from a usual vector of size $p$ is that the different values, that will be called local, are blind from each other: it is only possible to access the local value $x_i$ in two cases

- locally, on processor $i$ (by the use of a specific primitive)
- after some communications

These restrictions are inherent to distributed memory parallelism; here they are enforced by the use of an opaque type. This choice also makes parallelism fully explicit and we think programs more readable and BSP costs easier to analysis [19]. Worth noting is that parallel vectors, as any other type, can be embedded in higher-order types or used by polymorphic functions, e.g. mapping a function on a list of parallel vectors is done with List.map from the standard library. They can not, however, be embedded in themselves since the BSP machine has only one level of parallelism.

Since a BSML program deals with a whole parallel machine and individual processors at the same time, a distinction between the levels of execution that take place will be needed (see right scheme in Fig. 2.1):

- **Replicated** execution is the default. Code that does not involve BSML primitives (nor, as a consequence, parallel vectors) is run by the parallel machine as it would be by a single processor. Internally, replicated code is executed at the same time by every processor, and leads to the same result everywhere. The user, though, does not need to know this and can imagine a single, sequential execution.

- **Local** execution is what happens inside parallel vectors, on each of their components: the processor uses its local data to do computation that may be different from the others. Replicated and Local execution are strictly disjoint, and typically, processors alternate between them.

- **Global** execution concerns the set of all processors together, but as a whole and not as a single processor like in replicated execution. Typical example is the use of communication primitives, but the manipulation of vectors as a data structure, without use of the primitives, is also global.

The distinction between local and replicated is strict. The fact that parallel vectors are an opaque structure prevents their contents from being read in replicated mode, which forms the base of the BSML model. Hence, the replicated code can not depend on local information, and remains replicated.
Parallel vectors are handled through the use of five different primitives that constitute the core of BSML:

- two asynchronous ones, one corresponding to the creation of parallel vectors and one to local accesses to values inside each member of the vector;
- two synchronous ones that perform communications;
- and one to parallel composition of BSP computations.

Implementation of these primitives rely either on MPI, PUB [7] or on the TCP/IP functions provided by the Unix module of OCaml. A toplevel is also provided where user can defined its number of processors: execution on the sequential toplevel or on a real parallel machine would give the same results (except in time).

Fig. 2.2 resumes the use of the primitives. Informally, primitives works as follow.

Primitive `mkpar` is the normal way of creating a parallel vector. It builds this parallel vector by applied the function passed as argument to the pid of every processor. Thus the function, is executed locally with different parameters as classical data-parallel computing.

Primitive `apply` is the primitive of local computation. At each processor, it applies a local function to a local argument.

Primitive `proj` is the dual of `mkpar`, and the only way to extract a non-parallel value from a parallel vector. Given a parallel vector, it returns a function such that, applied to the pid of a processor, it returns the value of the vector at this processor. `proj` performs communications to make local results available globally within the returned function. Hence it establishes a meeting point for all processors and, in BSP terms, ends the current super-step.

Note that the choice of functions (int → 'a) in `mkpar` and `proj`. Arrays of size p or lists could have been chosen instead, but the interface is more functional and generic this way. Furthermore, as seen in the examples, the conversion between one style and the other is easy. Internally, our implementation rely on arrays.

Primitive `put` is the comprehensive communication primitive. It allows any local value to be transferred to any other processor. As such, it is more flexible than `proj`. It is as well synchronous, and ends the current super-step.

The parameter of `put` is a vector that, at each processor, holds a function of type (int → 'a) returning the data to be sent to processor `i` when applied to `i`. Canonical use of `put` is

\[
\text{put} \left( \text{apply} \left( \text{mkpar} \left( \text{fun} \text{sendfrom} x \text{sendto} \rightarrow (\text{sendfrom},\text{sendto},x) \right) \right) \right)
\]

where expression `e` computes (or usually, selects) the data that should be sent depending on sender to sendto. The return of `put` is another vector of functions that return, when applied to `i`, the value received from processor `i`.

`put` is the most difficult primitive to handle, but it can handle any communication scheme: it is used when complex communication is needed, like when rebalancing data between the processors. Another way to understand `put` is to view it as a transposition operation on matrices if we consider the different processors as the columns:

\[
\begin{align*}
&f_0 & f_1 & \cdots & f_{p-1} \\
&f_0 & f_1 & \cdots & f_{p-1} \\
&\vdots & \vdots & \ddots & \vdots \\
&f_0 (p-1) & f_1 (p-1) & \cdots & f_{p-1} (p-1)
\end{align*}
\]

\[
\begin{align*}
&f_0 & f_1 & \cdots & f_{p-1} \\
&f_0 & f_1 & \cdots & f_{p-1} \\
&\vdots & \vdots & \ddots & \vdots \\
&f_0 (p-1) & f_1 (p-1) & \cdots & f_{p-1} (p-1)
\end{align*}
\]

<table>
<thead>
<tr>
<th>primitive</th>
<th>type</th>
<th>informal description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mkpar</code></td>
<td>(int → 'a) → 'a par</td>
<td>( f \mapsto (f 0, \ldots, f (p - 1)) )</td>
</tr>
<tr>
<td><code>apply</code></td>
<td>(a → 'b) par → 'a par → 'b par</td>
<td>( (f_0, \ldots, f_{p-1}) \mapsto (f_0 x_0, \ldots, f_{p-1} x_{p-1}) )</td>
</tr>
<tr>
<td><code>proj</code></td>
<td>'a par → (int → 'a)</td>
<td>( (x_0, \ldots, x_{p-1}) \mapsto (\text{fun} i \mapsto x_i) )</td>
</tr>
<tr>
<td><code>put</code></td>
<td>(int → 'a) par → (int → 'a) par</td>
<td>( (f_0, \ldots, f_{p-1}) \mapsto ((\text{fun} i \mapsto f_i 0), \ldots, (\text{fun} i \mapsto f_i (p - 1))) )</td>
</tr>
<tr>
<td><code>super</code></td>
<td>(unit → 'a) → (unit → 'a) → 'a + 'b</td>
<td>( f_a \mapsto f_b \mapsto (f_a (), f_b ()) )</td>
</tr>
</tbody>
</table>
2.2.4 Other features

BSP paradigm’s simplicity and elegance comes at a cost: the ability to synchronise a subset of the processors would break the BSP cost model. Subset synchronisation is used to recursively decompose computations into independent tasks (this is the divide-and-conquer paradigm). However, [39] proposes a natural way to fit divide-and-conquer algorithms into the BSP framework without using subset synchronisation and by using sequentially interleaved threads of BSP computation, called super-threads. An adaptation of this method to BSML was proposed in [33]: the parallel superposition.

This last primitive called super allows the evaluation of two BSML expressions. From the programmer’s point of view, the semantics of the superposition is the same as pairing but of course the evaluation of super $E_1, E_2$ is different from the evaluation of $(E_1, E_2)$. 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 $E_2$. The messages are obtained by concatenation of the messages 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). That do not interact with our purpose and we thus refer to [16][19][20] for more details (semantics, examples and implementations).

BSML could also be used to manage external memory (files system) [17]. This feature is necessary to have safe accesses to local files (on each processor) and global ones (available to all processors as in MPI-I/O). Local files could not be accessed in replicated environment and vice-versa for global files. We refer to [17] for more details about this library.

[23] presents how to managed exceptions in BSML. We extends this work in this article.

2.3 Examples

Utility functions

The primitives described in the previous section constitute the core of BSML. In this section, we define some useful functions for BSP computing where somes are given as additional libraries for BSML.

The mkpar primitive is often used to split a dataset among the processors. The following function is an example that selects a part of a list on every processor:

(* select_list: 'a list -> 'a list par *)

let select_list l =
  let len = List.length l in
  mkpar (fun i -> cut_list l (i * len / bsip, p) ((i + 1) * len / bsip, p))

where cut_list l a b returns the sub-list of the elements of l from index a (inclusive) to index b (exclusive).

apply is often used for all in-place operation on vectors. For example, keeping only a single value in a vector (e.g. before communications as broadcasting) is achieved with:
(∗ filter_pid: ‘a par → int → ‘a option par∗)
let filter_pid v i =
  let filter = mkpar (fun j → if i = j then fun x → Some x else fun x → None) in
  apply filter v

Most frequently, the same function is applied to the members of the parallel vector, in a data-parallel fashion. This is easily done with parfun:

(∗ replicate: ‘a → ‘a par∗)
let replicate e = mkpar (fun _ → e)

(∗ parfun: (‘a → ‘b) → ‘a par → ‘b par∗)
let parfun f v = apply (replicate f) v

To parallel map a function on a list (classical data-parallel skeleton [1, 9]) scattered as above is as simple as:

(∗ parmap: (‘a → ‘b) → ‘a list par → ‘b list par∗)
let parmap f parlist = parfun (List.map f) parlist

The proj primitive is often used at the end of a parallel computation to gather the computed results. For example, converting a parallel vector into a list is sometimes convenient:

(∗ proj_list: ‘a par → ‘a list∗)
let proj_list v = List.map (proj v) procs_list

where procs_list is the list of pids: [0; 1; . . . p − 1].

Parallel prefix computation

As a generalisation of the above, a simple one-step reduce (having $\oplus_{k=0}^{p-1} v_k$ one each processors from the parallel vector $(v_0, v_1, . . . , v_{p−1})$) could be done with:

(∗ simple_reduce: (‘a → ‘a → ‘a) → ‘a par → ‘a
let simple_reduce op v =
  let vl = proj_list v in
  List.fold_left op (List.hd vl) (List.tl vl)

The above reduce does not make use of parallelism. If the combination operator has some cost, we may prefer to reduce in a multi-step manner (a classical logarithmic way), doing the combinations locally. It is based on the classical parallel prefix computation:

$$\text{reduce } e \oplus \begin{bmatrix} v_0 & \cdots & v_{p-1} \end{bmatrix} = \begin{bmatrix} e & v_0 \oplus v_1 & \cdots & \oplus_{k=0}^{p-1} v_k \end{bmatrix}$$

This algorithm combines the values of processors $i$ and $i + 2^n$ at processor $i$ for every step $n$ from 0 to $\lceil \log_2 p \rceil$. Fig. 2.4 (left) gives the code.

The program (reduce’ 1), given an associative operator op ($\oplus$) and a neutral element e, gathers data at every even processor, then at multiples of 4, 8, etc. First step is the communication: the argument of put returns the communication unit unless sending from processor $(2 \times \text{step} + 1) \times i$ to $2 \times \text{step} \times i$ for any $i$. Then, the combination is done at processor $2 \times \text{step} \times i$ using op. At the last step, the full reduction is on processor 0.

The program could seem difficult to read. In particular, defining the parameter of put and nested calls to apply do not make its structure obvious. This is partly solved with the new syntax, goal of this article.

In the same manner, the parallel prefix computation can be done using 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 broadcasting 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.

Fig. 2.7 presents some experiments on a cluster with 10 Pentium IV nodes (with 1 Go of main memory per node) interconnected with a Gigabyte Ethernet network. The binary version is the one presented in Fig. 2.4 and version using the superposition is the one from Fig. 2.5. Both need $\log(p)$ super-steps. The values were arrays of floats representing polynomials and the binary operation is the sum of two polynomials. Diagrams show the average of the results with increasing size of polynomials.
Parallel sorting algorithm

Our last example is the sampling sort algorithm (PSRS) of Schaeffer in its BSP version [38]. The PSRS algorithm proceeds as follows. First, the lists of the parallel vectors (we assume that their lengths are \( \geq p^3 \)) are sorted independently with a sequential sort algorithm. The problem now consists of merging the \( p \) sorted lists. Each process selects from its list \( p + 1 \) elements for the primary sample and there is a total exchange of these values. In the second super-step, each process reads the \( p \times (p + 1) \) primary samples, sorts them and selects \( p \) secondary samples. In the third super-step, each processor picks a secondary block and gathers elements that do belong to the assigned secondary block. In order to do this, each processor \( i \) sends to processor \( j \) all its elements that may intersect with the assigned secondary blocks of processor \( j \).

Fig. 2.6 (left) gives the code using the current syntax where “vlengths” is the vector of list sizes, “extract_n n” extract \( n-1 \) elements evenly distributed; “slice_p” transform a sorted list to a list of list depending of the samplings and “p_merge” merges sorted lists. Fig 2.8 shows the results for a 20 dual-core cluster (Intel E2180 at 2GHz with 2Gb of memory) interconnected by a Gigabyte Ethernet network. Note that these benchmarks were done using the new syntax presented in next chapter.
let reduce = fun op neutral v =
  if step >= neutral then v else
  let comm = put (mkpar (fun vec v ->
    let rec reduce' step op neutral v =
      if step >= neutral then v else
      let super =
        fun vec =
          let _ = mkpar (vec) in
          match vec with
          | Some v -> v
          | None -> v
      in
      match vec with
      | Some v -> v
      | None -> v
    in
    reduce' (step + 1) op neutral v
    ) (v)
  in
  in
  reduce' (step + 1) op neutral v

Figure 2.4: Parallel reduce in past (left) and the same using the new syntax (right)

(* reduce*: int -> 'a list -> 'a list)
let rec reduce' =
  if step >= neutral then v else
  let comm = put (mkpar (fun vec v ->
    let rec reduce' step op neutral v =
      if step >= neutral then v else
      let super =
        fun vec =
          let _ = mkpar (vec) in
          match vec with
          | Some v -> v
          | None -> v
      in
      match vec with
      | Some v -> v
      | None -> v
    in
    reduce' (step + 1) op neutral v
    ) (v)
  in
  in
  reduce' (step + 1) op neutral v

Figure 2.5: Divide-and-conquer reduce computation (left) and the same using the new syntax (right)

Figure 2.6: Parallel sampling sorting in past (left) and new (right) syntax
Figure 2.7: Benchmarking of reducing algorithms (addition of polynomials)

Figure 2.8: Benchmarking of the PSRS sorting of lists.
Chapter 3

New Syntax

3.1 Main problem

Having a very small core of parallel operations is a great strength for the formalisation of the language. It makes the definitions clear and the proofs shorter. Being able to embed these primitives in higher-order functions is precious and allows seamless complex parallel operations in little code. However, the program, even if high-level, still has to deal with replicated values and parallel vectors, and the use of the primitives can sometimes become awkward. Indeed, every operation inside of parallel vectors has to call a primitive and define a function. This gets worse when working with multiple vectors, with nested calls to \texttt{apply}. Simply transforming a pair of vectors into a vector of pairs is written:

\begin{verbatim}
let combine_vectors (v,w) = apply (parfun (fun v w → v,w) v) w
\end{verbatim}

which is unsatisfactory because we have to define, each time, a specific anonymous function. This implies creating named parameters although our function will only be applied to our vectors, and can be confusing. Additionally, the programmer has to make the choice of keeping the name of the vectors for the parameters, giving the same name to values of different types, or give a new name which may be even more confusing.

Another view for the definition of BSML is to start with the notion of execution levels: one can declare code that will be executed globally, as in standard OCaml, and code that will be executed locally, and form a parallel vector. An interesting insight is that in BSML, when within a local section (\textit{e.g.} the body of a function passed to \texttt{mkpar}), all replicated values can be accessed, but any parallel vector still needs to be “opened” using \texttt{apply}. Let’s illustrate this:

\begin{verbatim}
let f (g:int → int) (x:int) (y:int) = mkpar (fun i → g (i*x/y))
\end{verbatim}

\texttt{g}, \texttt{x} and \texttt{y} are all replicated values, used locally. If however \texttt{x} was a vector and what was intended was to apply our operation to its components – and accessing local data in a local section seems a natural thing to do – we would have to add a new parameter to the function, and embed the expression in an \texttt{apply}. Even worse if the same is true for \texttt{y}: opening vectors is too cumbersome, and this is because it can only be done at the global level:

\begin{verbatim}
let f (g:int → int) (x:int par) (y:int par) =
    apply (apply (mkpar (fun i x y → g (i*x/y))) x) y
\end{verbatim}

that is hard to read.

3.2 General description

To not have this burden of writing, we define a new syntax for parallel vectors.
### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Type</th>
<th>Informal Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \llangle e \rrangle )</td>
<td>( t \par ) (if ( e ): ( t )) ( (e, \ldots, e) )</td>
<td>( i ) on processor ( i )</td>
</tr>
<tr>
<td>( \text{Spid} ) (within a local section)</td>
<td>int</td>
<td>( v ) on processor ( i ) (if ( v = (v_0, \ldots, v_{p-1}) ))</td>
</tr>
</tbody>
</table>

\[ \text{proj, put and super are unchanged, mkpar and apply are not needed anymore.} \]

Figure 3.1: Summary of BSML alternate syntax

Let \( \llangle x \rrangle \) be the vector holding \( x \) everywhere (on each processor). The \( \llangle \rrangle \) indicate that we enter a local section, and pass to the local level. Resulting, on the outside, in a parallel vector, replicated information is available inside the vector, as with the \textit{mkpar} above. Now, how to access local information? We add the syntax \( $$ \) to open the vector \( x \) and get the local value it contains. Fig. 3.1 resumes the use of the primitives.

Note that \( $$ \) can obviously be used only within local sections. It is dependant of the call-by-value strategy of OCaml: \( $$ \) is apply to a variable that thus been a full evaluated vector at this point that is a vector of values. It could not performs communications and barriers: we recall that nested of vectors or evaluation of global primitives is forbidden inside parallel vector in BSML; we refer to [14, 22] for more details and a type system that prevent programmers to this. It is now possible to write our examples:

\begin{verbatim}
let combine_vectors (v,w) = \llangle v, w \rrangle
let f (g:int \rightarrow int) (x:int par) (y:int par) = \llangle g (i * x / y) \rrangle
which is shorter, more clear and thus less error-prone. Additionally, the local pid can be accessed with \( \text{Spid} \), to replace calls to \textit{mkpar}.
\end{verbatim}

Note that the synchronous primitives and superposition do not need a special syntax (they are not nested often like \textit{apply}), but their use is already made more simple. For example, the canonical scheme of collective communications:

\begin{verbatim}
put (apply (mkpar (fun sendfrom x sendto -> e(sendfrom,sendto,x))) x)
\end{verbatim}

which computes parallel vector of values \( x \) to be communicated depending on source and destination. Now it can be written:

\begin{verbatim}
put \llangle fun sendto -> e(Spid, sendto, $x$) \rrangle
\end{verbatim}

We thinks that is easier to read than the above code.

For example, the filter function of previous section is difficult to read, because the reader has to manually trace what the indices \( i, j \) refer to. The alternate syntax version below is much more obvious:

\begin{verbatim}
let filter_pid $v$ $i$ = \llangle if $\text{Spid} = i$ then Some $v$ else None \rrangle
\end{verbatim}

### 3.3 Implementation

Our new syntax is equivalent to the initial one. Indeed, \( \llangle \rrangle \) combined with \( \text{Spid} $$ \) is strictly equivalent to \textit{mkpar}, and the use of any other vector in $$ is equivalent to a call to \textit{apply}. An automated syntactic transformation modifies the code using the local section delimiters into code using the primitives. The transformation is implemented in the language using the generic OCaml pre-processor and pretty-printer Camlp4, and programs can be written using both local sections and primitives.

The transformation proceeds as follows:

- the contents of a local section \( \llangle e \rrangle \) is turned into the body of a function \( (\text{fun} \ldots \rightarrow e) \)
- This function is built automatically, every $$ in the local section being turned into a function argument.
- \textit{mkpar} is applied to that function, with its argument corresponding to \text{Spid} $$ .
- The appropriate \textit{apply} are then put around it to match the remaining function arguments to their corresponding parallel vectors.
On the other way, turning the initial syntax into this one would involve turning uses of `mkpar` into local sections. If the parameter $f$ of `mkpar` is a value, it just needs to be applied to $pid$ (possibly with $\beta$-reduction if we want this transformation to be the inverse of the previous one). Otherwise, it needs to be reduced to a value first by means of \texttt{let x = f in \ll \ldots \gg}, which indeed makes the parts that are reduced in parallel more obvious in the new syntax. In the same way, the transformation of calls to `apply` into local sections are straightforward as long as we deal with values.

In details, the rewriting of local sections \ll \ldots \gg in the code to the use of the primitives were done using the “quotations” of Camlp4. It works as follows: when entering a local section \ll e \gg, a reference is created to contain the list of its parameters and the transformation begins. At the analysis of his body, an “antiquote” \textit{i.e.} syntax $x$, will give to a special treatment: it creates a fresh variable “fresh_id\footnote{Simulated by the prefix \_\_Li\_ Camlp4 has no real mechanism for fresh variable.}” that will replace the $x$, and add to the list of parameters the pair (fresh_id, x). In the case where x is a simple identifier, the transformation verifies that it is not already in the parameters, and reuses the variable corresponding to where necessary to avoid nested of a `apply` for instance of the same variable.

Once the body “e” of the local section analysis in this manner, the transformation just turns it into the following BSML term: `mkpar (fun fresh_id_this \to e)`, where “fresh_id_pid” is the fresh variable was attributed to pid and then extend the call to `apply` by recursion on the list of parameters. This last step is performed by the function:

\begin{verbatim}
let rec build_apply _loc e explist = match explist with 
[| [] \to e 
| ex::exr \to \langle expr < apply $build_apply _loc e exr$ $ex$ \rangle

where the syntax \langle expr \ll \ldots \gg to build an node in the syntax tree. This entire transformation is less than a hundred lines of code using Camlp4.

We witnessed an important reduction in the length of code we translated from primitives-based to local sections-based, and believe this syntax makes BSML easier to read and understand.

### 3.4 Disadvantages of this new syntax ?

In general term, the sequence of super-steps and the deterministic nature of the BSP model, make the use of classical formal methods easier. In fact, developing this kind of software is tedious and we believe that just adapting formal method for parallel programs at the top of sequential ones is the best way to quickly provide useful tools.

For example, OCaml, is one of the possible targets for program extraction in the Coq proof-assistant \cite{four}. This will enable us to extract proved BSML programs from their proved Coq definitions \cite{15,22} by axiomatizing BSML primitives in Coq.

Using a specific syntax makes this work impossible in practice: ones should modify the type analyser of Coq which is a evil work. But past BSML primitives could be easily coded using this new syntax making compatible old programs notably certified ones which are extracted from Coq \cite{15,22}:

\begin{verbatim}
let replicate e = \ll e \gg
let apply vf vv = \ll vf vv \gg
\end{verbatim}

It is thus possible to have twice: primitives for proofs and a simple syntax for programmer, both are equivalents.

Readers could point out that the $x$ construction works only on variable. It is natural to think that it could also work on “values”. For example, having

\begin{verbatim}
let combine_vectors (v,w) = \ll (v,w) \gg
\end{verbatim}

that is $e$ would inductively transform each parallel vector of “e” into a local value. It is possible and easy to implement because in the parallel (distributed) implementation of BSML (using MPI, PUB, TCP/IP etc.), a parallel vector is just the local value on each processor \cite{16,32}: type ’a par = ’a. In this way, “$e$” would just generated the OCaml code “e”.

But that complicated analysis of the BSML program. What is the ML type of “$e$”? For example:

\begin{verbatim}
(* destruct: ’a \to ’b par ? but which ’b ? *)
let destruct v = \ll v \gg
\end{verbatim}

\footnotetext{Simulated by the prefix \_\_Li\_ Camlp4 has no real mechanism for fresh variable.}
For tuple, list etc. it is easy but not for more complex data structures.

Indeed, it could be confused programmers: can it put any expression inside “Se$” ? Clearly not, due to the structural nature of BSML (nested of vector is forbidden). But some $\eta$-extensions could solve this problem. Nevertheless, we think that the restriction to variables is sufficient in many cases and some $\eta$-extensions could be used elsewhere. We believe that writing these specific parallel codes inside a parallel vector and rejecting others is too difficult to understand and make the reading of the code too hard.

3.5 Small examples

Note that for BSML programs that are just a combination of functions from the standard library that perform communications, the new syntax does not bring back anything. The new syntax clearly outperforms the old ones when the programmer needs to code new patterns of communications and when vectors and the pid is used many times.

Thus, in these followings examples, the new syntax is not really easier to read on all the code. But in the worst case, the reading does not complicate.

3.5.1 Parallel Sampling Sort Algorithm

Fig. 2.6 (right part) presents the code for a parallel sorting algorithm [38] using the two syntax. Both are equivalents in size but using the new syntax, the code seems easier to read (we thinks so) because the code combine only pre-defined function of the BSML library that are really needed (the function doing a total exchange of the samples). Thus, the simple and barbing functions that were used in the past syntax are now not present in the new code.

3.5.2 Parallel reducing

Another more complete example is the parallel reduce program of Fig. 2.4 rewritten using the new syntax (right part). The program is both shorter and easier to read, with less inline functions and parameters cluttering the view. Conditions on the current pid (if $\$pid\$...) are read as thus easily, while places where real computation happens are more obvious. There is only one inline function, and its parameter appears next to put, making clear that it is the pid of the processor to send to.

It the same things that happen in the code of the parallel reducing of Fig 2.5. The code using the new syntax is clearly easier to read due to the explicit construction of the parallel vector and explicit calls to past computed vectors (and pid).

3.5.3 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 $\sqrt{n}$.

Fig. 3.2 gives the BSML code of the 3 methods. We used the following functions: “elim:int list $\rightarrow$ int $\rightarrow$ int list” which deletes from a list all the integers multiple of the given parameter; “final_elim:int list $\rightarrow$ int list $\rightarrow$ int list” iterates “elim”; “seq_generate:int $\rightarrow$ int $\rightarrow$ int list” which returns the list of integers between 2 bounds; and “select:int $\rightarrow$ int list $\rightarrow$ int list” which gives the $\sqrt{n}$th first prime numbers of a list. It is an example where most of the codes are applications of the same function on each processor (pure data-parallelism) and thus the new syntax is a little help for the reading but with less inline functions and parameters cluttering the view.

Logarithmic reduce method.

For our first method we use the classical parallel prefix computation. 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 $i \times \frac{n}{P} + 1$ and $(i + 1) \times \frac{n}{P}$. Each processor computes a local sieve (the processor 0 contains thus the first prime numbers) and then our scan is
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 an 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 eliminate 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.

Fig. 3.2 gives the measured performances for a cluster with 10 Pentium IV nodes (with 1 Go of main memory per node) interconnected with a Gigabyte Ethernet network. Note that we obtain a super-linear acceleration for 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.
3.5.4 The \( N \)-body problem

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

\[
E = - \sum_{i=1 \atop i \neq j}^{N} \sum_{j=1}^{N} \frac{m_i \times m_j}{r_i - r_j}
\]

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\(^1\). 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 \( N/p \) point masses.

Fig. 3.3 gives the BSML code of the 2 algorithms using the two syntax: past is the right and newer in the left (shared sequential code is at the top). We use the pair_energy function that 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.

Systolic loop

Our second algorithm is based on a systolic loop [28]. 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. Fig. 3.5 presents the code using the past and new syntax. Note that using the new syntax, we can naturally defined operator “op” in a pure sequential manner which is easier to understand than in the past syntax. The systolic loop uses the “shift_right” function which shifts the values from each processor to its right-hand neighbour. Fig. 3.6 presents the code using the past (left) and new syntax (right) for this new pattern of communication. Once more, the code using the new syntax is easier to read.

For the \( N \)-body problem, 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 \( p - 1 \) 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: \[
\begin{align*}
& p \times \left( \frac{N}{p} \times g + L + 2 \times \frac{N}{p} + \frac{N}{p} \times \frac{N}{p} \right) + p \times g + L \\
\equiv & \ N \times g + p \times L + 2 \times N + \frac{N}{p} \times N + l + p \times g + L
\end{align*}
\]
that is the same as before but with more synchronization time.

Fig. 3.8 gives the measured performance on a cluster with 10 Pentium IV nodes (with 1 Go of main memory per node) interconnected with a Gigabyte Ethernet network. 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 particles 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 [17].

\(^1\)There exist more sophisticated algorithms that take advantage of the symmetry of the sum but this is not the subject of this article.
Definition and sequential part

\[ \text{let } \text{resp. } B \text{ round-robin distribution of the blocks. This algorithm has the following BSP cost [21]:} \]

\[ \text{let } \text{minus_point } (x_1,y_1,z_1) (x_2,y_2,z_2) = (x_1 - x_2, y_1 - y_2, z_1 - z_2) \]

\[ \text{let } \text{length_point } (x,y,z) = \sqrt{x^2 + y^2 + z^2} \]

val pair_energy : atom list \rightarrow \text{atom list}\]

\[ \text{let } \text{pair_energy some_bodies other_bodies =} \]

\[ \text{List.fold_left } (\text{fun energy } \rightarrow \text{function } (r1,m1)) \rightarrow \]

\[ \text{energy*} / \text{List.fold_left } (\text{fun energy } \rightarrow \text{function } (r2,m2)) \rightarrow \]

\[ \text{let } \text{r=length_point } (\text{minus_point } r2 r1) \text{ in}\]

\[ \text{if } r > 0 \text{ then energy} += (\text{m1} * \text{m2})/r \text{ else energy}\]

\[ \text{0. other_bodies}) \text{ in}\]

\[ \text{0. some_bodies} \]

Total exchange method

\[ \text{let final_ex = parfun2 pair_energy my_bodies energy in}\]

\[ \text{let res_final= simple_reduce (+.) 0. final_ex in}\]

Systolic method

\[ \text{let energy=parfun2 pair_energy my_bodies my_bodies in}\]

\[ \text{let final_sys = systolic pair_energy (+.) my_bodies energy in}\]

\[ \text{let res_final= simple_reduce (+.) 0. final_sys in}\]

\[ \text{...} \]

Figure 3.3: BSML code for the N-body problem (left) and the same using the new syntax (right)

3.5.5 Dense Matrix Multiplication

Interest of the multiplication of matrices is not more to present: methods to solve this problem are numerous and parallelism is one of the possible answers. Our implementation is based on an algorithm presented in [21] which is independent of type of data of the matrices (at the difference of algorithms of [38]). Also, we consider dense matrices (algorithms for sparse vector-matrices BSP multiplications are presented in [5]).

Initially, two \( n \times n \) matrices \( A \) and \( B \) are distributed among the \( p = \sqrt{p} \times \sqrt{p} \) processors so that each processor stores a sub-set (call block) of size \( m \times m \) (where \( m = \frac{\sqrt{p}}{\sqrt{p}} \)) of the original matrix. In this manner, element \( A(i, j) \) (resp. \( B(i, j) \)) with \( 0 \leq i, j < n \) is stored in the \( \left( \frac{L}{m} \right) \times \sqrt{p} + \frac{j}{m} \)-th block. We call \( A_i \) (resp. \( B_i \)) the \( i \)-th block of \( A \) (resp. \( B \)) shared by processor \( i \). We note \([0]\) an empty matrix and \( \oplus \) (resp. \( \otimes \)) sum (resp. multiplication) of matrices. The algorithm can be written as follow:

\[ \text{begin Mult}(C,A,B) \]

\[ \text{let } m = \frac{\sqrt{p}}{\sqrt{p}} \]

\[ \text{let } p_i = \text{pid} \mod \sqrt{p} \text{ and } p_j = \text{pid} \mod \sqrt{p} \text{ and } C_q = [0] \text{ in}\]

\[ \text{for } 0 \leq l < \sqrt{p} \text{ do}\]

\[ \text{begin}\]

\[ \text{let } a = A_{((p_i + p_j + l) \mod \sqrt{p}) \times \sqrt{p} + p_i} \]

\[ \text{and } b = B_{((p_i + p_j + l) \mod \sqrt{p}) \times \sqrt{p} + p_j} \mod \sqrt{p} \]

\[ C_{p_i p_j} \leftarrow C_{p_i p_j} \oplus a \otimes b \]

\[ \text{end}\]

\[ \text{end Mult}\]

Figures can remark that each processor received data from two distinct processors at each super-step due to a round-robin distribution of the blocks. This algorithms has the following BSP cost [21]:

\[ \sqrt{p} \times \left( \frac{(2n - 1)n^2}{p^2} + \frac{2n^2}{P} \times g + L \right) \]

For our BSML implementation, sequential multiplication and sum of matrices are naive ones. \( e \) is a neutral element. We represented matrices as arrays as [21] in C. We also used a utility function “get_from” which allow to gets data from processors to other processors (pattern of communication) depending of a parameter (Fig. 3.3).

Fig. 3.3 give a generic implementation of this algorithm using BSML. Except for the “get_from” function, the new syntax is not useful. This mainly due to the fact that most of the code is dedicated to sequential computations.
let multiply_par e n mult plus parA parB parC = (⋯) in
let n = n / sqrt_p in
(⋯)
let pids_to_sendA l pid = (⋯) in
let pids_to_sendB l pid = (⋯) in
for l = 0 to sqrt_p - 1 do (⋯) done

Figure 3.4: BSML code of the matrix multiplication (left) and the same using the new syntax (right)

Fig. 3.10 presents some experiments on a cluster with 10 Pentium IV nodes (with 1 Go of main memory per node) interconnected with a Gigabyte Ethernet network. Each matrix contains complex numbers (record of two floats). Diagrams show the average of the results with increasing size of matrices.
val systolic : ('a -> 'b) -> ('c -> 'b par -> 'c) -> 'a par -> 'c par -> 'c

let systolic f op vec init =
  let rec calc n v res =
    if n = 0 then res else
      let newv = shift_right v in
      calc (n - 1) newv (op res (parfun2 f vec newv))
  in calc (bsp_p()) vec init

val systolic : ('a -> 'b) -> ('c -> 'b par -> 'c) -> 'a par -> 'c par -> 'c

let systolic f op vec init =
  let rec calc n v res =
    if n = 0 then res else
      let newv = shift_right v in
      calc (n - 1) newv (f vec newv)
  in calc (bsp_p()) vec init

Figure 3.5: Generic systolic loop (left) and the same using the new syntax (right)

val shift: int -> 'a par -> 'a par

let shift dec datas =
  let p = bsp_p() in
  let mkmsg = fun pid data dst ->
    if dst = (dec + pid) mod p then Some data
    else None
  in
  apply (parfun (compose noSome) (put (apply (mkpar mkmsg) datas)))

val shift: int -> 'a par -> 'a par

let shift dec data =
  let comm = put (fun dst ->
    if dst = (dec + pid) mod p then Some data
    else None) in
  match comm (f pid) with
    | None -> failwith "Impossible case"
    | Some v -> v

Figure 3.6: Shifting values of processors (left) and the same using the new syntax (right)

val get_from : (int -> int) -> 'a par -> 'a par

let get_from f parv =
  let comms = put (apply (mkpar (fun pid v dst ->
    if pid = (f dst) then Some v
    else None) parv)) in
  apply (mkpar (fun rcv ->
    match rcv (f pid) with
      | None -> failwith "Impossible case"
      | Some v -> v)) comms

val get_from : (int -> int) -> 'a par -> 'a par

let get_from f parv =
  let comms = put (fun v dst ->
    if f pid = (dec + pid) mod p then Some data
    else None) in
  match comms (f pid) with
    | None -> failwith "Impossible case"
    | Some v -> v

Figure 3.7: Communications in the matrix multiplication (left) and the same using the new syntax (right)

N-body problem with Systolic and Total exchange methods

Figure 3.8: Performance of the N-body problem.
Figure 3.9: Performances of the sieve of Eratosthenes

Figure 3.10: Execution time of multiplications of dense matrices of complex
Chapter 4

Pattern matching of parallel vectors

Pattern matching is an important feature of high-level programming languages (OCaml, Java etc.) It offers sub-term extraction and definition of functions by cases. The problems with the matching of parallel values (as parallel vectors) are to define suitable patterns and to perform the matching (which requires communications) as efficiently as possible.

Patterns is also often be used to exception handler. For example, in OCaml, syntax for pattern-matching of data structures is essentially the same for catching exceptions.

A first work for pattern-matching for BSML were done in [11] but without implementation and for exceptions catching. Also, there were no proj [16] primitives which now allow a more readable code than these patterns.

We now present how the explicit notation of parallel vectors allows use interesting patterns for BSML.

4.1 General ideas

The main idea is to perform matching on a vector parallel. This matching is done globally, and is a global decision. indeed, it is allow and fruitful to use the standard matching of OCaml locally to take a local decision, or overall on a replicated value.

Matching allows you to search for values corresponding to a ground in a vector (or set of exceptions, see next section) without the order of the elements of the vector (or all) has a significance. It is introduced with a construction similar to match with of OCaml:

\[
\text{matchpar } e \text{ withpar } \\
\parallel \text{parallel_pattern1 } \rightarrow e1 \\
\parallel \text{parallel_pattern2 } \rightarrow e2 \\
\text{...}
\]

Fig 4.1 gives the BNF syntax of the parallel pattern matching. A parallel patterns, among \( \parallel \) is a list of patterns separate by \( \parallel \) which will be searched successively in the vector (resp. in a set for exceptions). These patterns can be either a standard OCaml pattern or a one with a star (pattern)\(^*\) for any number, strictly positive occurrences of this pattern in the vector (or the set for exceptions) or the default pattern _ for any content vector.

For example, the pattern \( \langle\langle (None)\rangle\rangle\) is a vector whose elements are all “None”, and \( \langle\langle \text{None} \parallel_\_\rangle\rangle\) a vector with at least one “None”.

As patterns in OCaml, a parallel pattern can bind variables: \( \langle\langle \text{Some } x \parallel_\_\rangle\rangle\) bind \( x \) in the first instance of the constructor “Some” found in the vector. The result is a bit more complex for repeated patterns: \( \langle\langle (\text{Some } x)\rangle\rangle\)\(^*\) \( \parallel_\_\rangle\rangle\) should link the variable \( x \) to a number of different values unknown \textit{a priori}. The behavior of the filtering is to allocate a list for all variables appearing in a star: “\( x \)” here is the list of the corresponding variables.

The filter works by elimination: in case of overlapping reasons, only first appearing in the list and accepting the matched term is considered. Thus, the pattern \( \langle\langle (0)\rangle\rangle\)\(^*\) \( \parallel (x)\rangle\rangle\) label to \( x \) the list of non-zero values of the vector, since any zero will be eliminated by the first pattern.

Finally, it is sometimes useful to know what processor assigned result. The syntax pattern “at p”, where “p” is a variable name, bound to “p” pid of processors where the pattern was consistent. In case of repeated motif, as for other variables, a list of correspondence is related to p. It is thus easy, using the functions of the OCaml’s standard library “List.combine” and “List.split” to work on a list of pid-value associations.
Figure 4.1: Abstract syntax of parallel patterns

For examples, ones can code a total exchange as:

\[
\begin{align*}
& (\ast \text{ total: } 'a \text{ par} \rightarrow 'a \text{ list}) \\
& \text{let total vl = matchpar vl withpar } \langle x \rangle \rightarrow x
\end{align*}
\]

In a similar way, ones can use the ability of pattern matching to test if all elements of a parallel vector satisfies a condition (here empty) and else computing something for which not:

\[
\begin{align*}
& \text{matchpar } \langle \text{is_empty } v \rangle \text{ withpar } \\
& | \langle \text{false at } p \rangle \langle \_ \rangle \rightarrow \ldots \\
& | \langle \_ \rangle \rightarrow \ldots
\end{align*}
\]

Without parallel pattern matching and new syntax, this kind of test is boring to code:

\[
\begin{align*}
& \text{let new_is_empty pid v = if (is_empty v) then [] else [pid]} \\
& \text{let op e1 e2 = match e1.e2 with} \\
& | [],[] \rightarrow [] \\
& | \text{hd::tl},[] \rightarrow \text{hd::tl} \\
& | [],\text{hd::tl} \rightarrow \text{hd::tl} \\
& | \_ \rightarrow \text{List.concat e1 e2} \\
& \text{let my_list = simple_reduce op (apply (mkpar (fun pid \rightarrow is_empty pid))) v} \\
& \text{if (List.empty my_list) then } \ldots \text{ else } \ldots
\end{align*}
\]

The filtering method has the advantage here to store information about the processors involved. It lets you add event more specific like “some data on a single processor” \( \langle \text{false at } p \rangle \langle \text{true} \rangle \ast \) or “all processors still have data” \( \langle \text{false} \rangle \ast \), cases that might be useful to balance the distribution of data for example. This kind of result codes only with “simple_reduce” would have required the definition of complex ad-hoc operators.

The implementation of standard functions such as “List.find”, “List.filter” or “List.assoc” of OCaml is also simplified for parallel lists obtained by a distribution such as on described in the previous section. This result is generalized to both functions on parallel sets or hash-tables [18]. For example, the “List.assoc” parallel function could be coded:

\[
\begin{align*}
& \text{let assoc_par x vl = } \\
& \text{let vassoc = } \langle \text{try Some (List.assoc x $vl$)} \text{ with Not_found } \rightarrow \text{None } \rangle \text{ withpar} \\
& \text{matchpar } vassoc \text{ withpar } \\
& | \langle \text{Some x \_} \rangle \rightarrow x \\
& | \langle \text{None} \rangle \rightarrow \text{raise Not_found}
\end{align*}
\]

The matching proposed here can be considered as syntactic sugar in the sense that it adds any functionality related to parallelism: the block of matching applied to a vector parallel \( v \) starts doing communications with \textbf{proj} for reduce the vector to a replicate list and performs the matching choice in a replicated manner.

### 4.2 Exceptions

We first briefly describes the problems of adding exceptions in BSML. More details could be find in [23][22].

#### 4.2.1 Problems

In OCaml, once an exception is raised, it is propagated until it meets an enclosing block:

\[
\begin{align*}
& \text{try } \ldots \text{ with } \text{pattern1 } \rightarrow \text{t1} | \text{pattern2 } \rightarrow \text{t2}
\end{align*}
\]
that pattern-matches against the exception. The exceptional behaviour 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 a problematic case.

When an exception is raised locally, but not caught immediately, the processor concerned is not going to execute any of the replicated code that might occur until the end of the super-step: 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 super-step and cause a deadlock when the other processors reach the barrier. For example:

```ocaml
let vs = if $pid$=0 then raise Failure else (fun _ → $pid$)
```

Here, an exception is raised locally on processor 0 but other processors continue to follow the main execution stream, until they are stopped by the need for a synchronisation. Then, a deadlock occurs.

The solution we provide intends to stay as familiar as possible to the ML programmer. We used a construct to catch globally exceptions that are raised locally. For our example:

```ocaml
trypar ≪ if $pid$=0 then raise Failure else (fun _ → $pid$)≫
```

At each `withpar`, a barrier occurs and local exceptions (raise locally that is inside a parallel vector) are replicated (gathered on each processors) making possible to treat them. 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.

As regards exceptions, all exceptions lifted locally has already been reduced at the global level at the time of screening.

### 4.2.2 Patterns for exceptions

In [23], each `withpar` have only one patterns : a variable that represents the set of locally raised exceptions. Now with our new patterns syntax, this set could be represented as a parallel vector: there were only one possible exception per processor since no local code could be executed if a processor have raise a local exception. We thus used the same syntax.

For example, the “assoc_par” function. In the previous code, in the current super-step, “List.assoc” returns a value of type option, because a local failure (“Not_found”) is normal (defined in the OCaml’s standard library) and must not transform into a total error. Now, we could write, assuming previously defined exception “Found”, a code only using catch up of local exceptions:

```ocaml
let assoc_par2 x vl =
  trypar ≪ raise (Found (List.assoc x $vl$))≫
  withpar
  ≪ Found y \ _ ≫ → y
  ≪ (Not_found)∗≫ → raise Not_found
```

The local expression raises the exception “Not_found” or “Found” if a result was found. The parallel patterns allow to quickly distinguish between these two cases, “a result was found” or “no processor has found a result”.

### 4.3 Implementation

Like our other syntactic extensions, it is implemented using Camlp4. The term to be matched is first projected into a list, then correspondence of the list in each of the parallel patterns is tested as follows: it eliminates successively from the list the elements corresponding to each of the standard patterns appearing in the parallel pattern. This raises a failure if no evidence was found. Note that the pattern _ absorbs the entire list.

Once all the patterns tested in this way, if the list remaining is empty, the pattern corresponds to the value. In the contrary case, the next pattern shall be considered.
let f x =
(* the initial argument is first projected into the form of a list of (pid,value) pairs *)
let __l = Bsml.proj_list_pids x
in
(* block try..with corresponding to the parallel pattern ≪ Some v at p \ (None) ≫ *)
try
let (v, p, __l) =
let rec __bsml_patt_aux acc =
(* Matching for (Some v at p) *)
function
| x :: r →
  (match x with
   | (p, Some v) → (v, p, (List.rev_append acc r))
   | x → __bsml_patt_aux (x :: acc) r)
| [] → raise Bsml.Parmatch_nextcase
in __bsml_patt_aux [] __l in
let (__doesmatch, __l) =
(* pattern (None) *)
List.fold_right
(fun x (__doesmatch, __l) →
match x with
| (_, None) → (true, __l)
| _ → (__doesmatch, (x :: __l)))
__l (false, [])
in
(* we verify that the entire has been well verified
and we apply the result (Some v) in this case *)
if not __doesmatch
then raise Bsml.Parmatch_nextcase
else if __l <> []
then raise Bsml.Parmatch_nextcase
else Some v
with
| Bsml.Parmatch_nextcase →
(* next pattern: parallel pattern ≪ _, _ ≫ *)
(try
  let __l = []
in if __l <> []
  then raise Bsml.Parmatch_nextcase
else None
(* fail of the parallel matching *)
with
| Bsml.Parmatch_nextcase → raise Bsml.Parmatch_failure)

Figure 4.2: Example of the parallel pattern transformation

The difficulty of implementing this mechanism in the form of an automatic rewriting is that the parallel patterns reuse standard OCaml patterns to matching elements of the parallel vector: they contain binding variables that can be a value or a list of values (star case). To do so, the transformation starts by extracting the list of these variables, then create a construction let (var1, ..., varn) = [parallel matching] in [expr] so that the variables are linked as desired.

Fig 4.2 presents an entire example of this transformation. Left part present the parallel pattern and right part the generated code. This code has the following structure:

1. It uses the function “proj_list_pids” to project the parallel vector to filter and get a association list (pid, value).

2. Different parallel patterns are chained together with exceptions: each block of the matching is placed in a try with structure, and raises the exception “Bsml.Parmatch_nextcase” in case of mismatch: the caught exception leads to the filtering block corresponding to the parallel pattern and so on to match or failure. In case of failure, a final caught of exception is added and raises the exception “Bsml.Parmatch_failure”, similar to the exception “Match_failure” of OCaml.

3. Within each parallel pattern, we built a function for each pattern: That gives a search function which is simply applied to the original list; for star pattern, we built a filtering function which be applied to using “List.fold_left”. In both cases, the function returns a tuple consisting of the variables bound by the pattern, and the list of non-matched patterns.

4. Once the end of the parallel pattern is reached, we test if the entire list has been filtered: if so, the pattern
is validated and evaluated the corresponding expression in a context with the bind variables by the pattern, otherwise the exception Bsml.Parmatch_nextcase refers to the following case.

Implementation of how local exceptions are managed *i.e.* gathered on all processors and transformed to a set of exception using the `trypar withpar` construct is not the subject of this article. Implementation issue could be find in [23][22]. Patterns for exceptions are implemented as other ones except that they used the set of local exceptions raised in the code.
Chapter 5

Full example

5.1 Parallel backtracking

5.1.1 Description of the problem

Search depth, or backtracking, is often used to solve problems of satisfaction of constraints: it is a generic method which consist to recursively sub-dividing the space of answers, explore all the opportunities that are likely to lead to a solution. There that usually used to solve NP problem, when no ad hoc solution is known.

Backtracking is generally represents as an in-depth exploration of the responses organized as a tree, cutting off any branch that will surely lead to no response. For example, satisfaisability a logical formula of $n$ boolean variables (SAT problem) can be analyzed by breaking down all the possibilities in binary tree of depth $n$, a node for which the set variables is able to reduce the formula to false can be used to cut the sub-trees that get out. If you reach a leaf, the formula is satisfied and the problem resolved.

Due to the exponential nature of the problems to solve, backtracking is quickly costly. For example, we propose here a generic implementation of backtracking in BSML. The challenge is to fairly allocate the nodes to be explored between the different processors and rebalance when necessary, the branching factor is not expected in the general case.

For this example, assume the problem provided in the form of the root of a tree and a “child” function that tests if a node is a solution (and raise an exception if it is) or returns his sons in the form of a list. The use of an exception allows to catch up the solution easily despite the nested of recursion, and we can show the use of parallel patterns. This is one case where the use of exceptions and to catch is is particularly useful for the programmer.

5.1.2 Sequential algorithm

A sequential solution could be coded as follow:

```ocaml
let rec down n =
  ignore (List.rev_map down (childs n))

let backtrack () =
  try
    down root; None
  with
  | Solution_found s → Some s
```

where the “down” function is responsible of the recursive calls for the exploration of the tree. It returns no result. It is the exception raise by the function “childs” that deal with it when necessary. We use the function of the OCaml’s standard library “List.rev_map” which is equivalent to “List.map” followed by “List.rev” but which is more efficient: for our application, the exploration order not matter.

5.1.3 Naive parallel version

The parallel version follows the same process, except that the nodes are distributed on the components of a parallel vector. This first naive solution would be:
let backtrack () =
  trypar
  let init = select_list (childs root)
  in ≪ down $init$≫ ; None
  withpar
    | ≪ Solution_found s \_ ≫ → Some s

Here, we begin by calculating an initial set of nodes (using “childs root”) that are distributed across the processors. We also suppose that the cardinal of the initial set of nodes is greater than \( p \) to simply have nodes on all processors. The descent is then performed sequentially on each processor. When one of them found a solution, it locally throw the exception “Solution_found” to be catch by the overall \( \text{trypar withpar} \) structure.

This simplistic approach has two main defects: it does not balancing, and therefore assumes that all initial nodes can be distributed fairly to all processors and the local sub-sets will result in a comparable amount of calculations. This is generally not the case, some branches being fasted, resulting to computations on a single processor.

Second defect, the program is fully asynchronous, and works in a single super-step: following our model of exceptions, a solution can be propagated at the end of this super-step and therefore once all processors have finished their calculations. The duration will be the same if a processor finds a solution quickly, and almost all of the tree will be explored in all cases. Note that this problem will be solved by doing a rebalancing that requiring barriers.

### 5.2 Efficient parallel version

#### 5.2.1 General method

A more advanced solution is to split the computation in super-steps, making them a part of exploration followed by an eventual rebalancing. Each rebalancing will be an opportunity to detect a possible solution. This cutting can be done in several ways: one of the simplest, is to maintaining a distributed set of nodes whose sons have not yet been explored and to authorize a maximal number of local operation for each super-step.

Our approach is slightly more stringent: it keeps the local calculations at the same level of depth in the tree. The rate of connection, although aleatory in general, depends on most of the depth. By exploring these levels one after another, we reduce the chances of a quick gap between the load of the processors, and thus limit the need for rebalancing. At each super-step, the parallel machine will explore a forest corresponding to sub-trees whose roots are subsets of nodes at level \( n \) in the tree to explore. This super-step ends when all the descendants of level \( n \) of these nodes have been calculated, which are rebalanced and partitioned, and giving rise to a new super-step. The parameter “nchilds” allows the width of the forest to explored for each super-step by each processor.

Fig. 5.1 gives the generic code of the parallel backtracking methods. The algorithm uses two main mutually recursive functions and has thus two levels of recursion. The first one, “down” is recursive relatively to the depth of the tree and starts to proceed by rebalancing the nodes. To do that, the function proceeds by first calculating the size of local lists and call the “rebalance_if_needed” function which takes the decision to rebalance based on this result, returning all the time a parallel list distributed fairly, and the total number of elements and the maximal local size). The second one,“explore” is responsible for the exploration in breadth: it recursively selects some of the nodes on each processor using “split_nodes” and carry on an in-depth exploration using “down”.

Each function returns “unit” when no more nodes need to be explored, leaving the caller to resume the exploration, which corresponds to the backtracking. The reader may be surprised by the fact that we impose a barrier at each level of descent in the tree. When applying the algorithm to a given problem, we compose several times the “childs” function so to get a connection rate sufficient for balancing. The frequency of barriers can be adjusted by the “nchild” parameter.

#### 5.2.2 The Rebalancing function

It remains to define a function able to rebalance the data. In our case, the list is not ordered, we thus calculate the number of desired elements on each processor, and move the excess of elements of a processor \( i \) to the first processor “cyclically to the left” of \( i \) (i.e. the first in the list \( [(i - 1) \mod p, \ldots, (i - p + 1) \mod p] \)) which has a deficiency number of elements.
let rec rev_split acc n = function
  | [] -> acc,[]
  | x:r -> if n = 0 then acc,x::r else rev_split (x::acc) (n-1) r

let rec down nodes = (* depth--recursive *)
matchpar List.length $nodes$ withpar
  | (0) -> () (* backtracking *)
  | (nl) ->
    let
      nodes, ntot, max_sz = rebalance_if_needed nodes nl
    in
      explore nodes max_sz
and explore nodes max_sz = (* width--recursive *)
let split_nodes =≪ rev_split [] nchld $nodes$≫ in
let
  current_nodes, next_nodes =≪ fst $split_nodes$≫,
 ≪ snd $split_nodes$≫
in
  down ≪ childs $current_nodes$≫;
if max_sz - nchld <= 0 then ()
else explore next_nodes (max_sz - nchld)

let backtrack () =
  trypar
  let
    init =≪ if $pid$=0 then childs root else []≫ in
  in
    down init; None
    withpar
      |≪ Solution_found s \ _≫ -> Some s

Figure 5.1: Parallel and generic backtracking methods using parallel patterns

let rebalance v sizes ntot =
  let deltas = List.map (fun (i,n) -> i, n - ((i + 1) * ntot / bsp_p - i * ntot / bsp_p)) (List.combine procs_list sizes) in
  let rec find_dest this size data nlost = function
    | (i,x)::r ->
      let
        delta = nlost + x
      in
      if delta >= 0 then find_dest this size data delta r
      else if size > -delta then
        let
          d1,d2 = rev_split [] (-delta) data in
          (i,d1)::find_dest this (size + delta) d2 0 r
        else
          let
            d1,d2 = rev_split [] size data in
            [ (i,d1); (this,d2) ]
in
  in
  let sendto_list =
   ≪ let left,(i,d)::right = rev_split [] $pid$ deltas in
      if d>0 then find_dest $pid$ d $v$ 0 (left @ List.rev right)
      else [ $pid$, $v$ ]≫ in
  let
    exch = put≪ fun sendto -> try List.assoc sendto $sendto_list$ with Not_found -> []≫ in
  in
    unordered_flatten (List.map $exch$ procs_list)

Figure 5.2: Code of the rebalancing function

Fig 5.2 presents the code of the rebalancing function. The function takes as parameters the distributed list “v”, the “sizes” list of length p involving local sizes of the lists on each processor, and the total size of the distributed list (the sum of “sizes”). The associations list “deltas” calculated for each processor, the gap in number of elements compared to the final situation. For a processor pid in excess of elements, the “find_dest” function compute how these elements should be allocated on other processors: it is recursive on the list of processor “cyclically to the left” of pid, and returns a list of associations (processor, number of elements). For simplicity, the non-displaced elements are regarded as sent in their original processor, which does not cause communications.

The communications are then calculated and performed by a call to put, and re-assembled as a distributed list by a local call to the function “unordered_flatten” (optimized version, un-ordered and of the OCaml standard “List.flatten”). It remains to determine a criterion of balance, to avoid unnecessary communication. Our criterion here is \( n_{max} - n_{min} > n_{tot} \frac{p}{p} + 1 \), which produces a balanced, while the differences between the amounts of local data on the total load is too large, but by allowing at least a gap of 1 for small sets. Fig 5.3 presents the OCaml
let rebalance_if_needed v sizes =
  let nmin,nmax,ntot =
    List.fold_left (fun (nmin,nmax,ntot) i        
    → min i nmin, max i nmax, i+ntot)
    (max_int,0,0) sizes
  in
  if nmax - nmin <= ntot / bsp_p + 1 then v,ntot,nmax
  else rebalance v sizes ntot,
  ntot,
  (ntot/bsp_p + if ntot mod bsp_p = 0 then 0 else 1)

Figure 5.3: Code of the “if needed” rebalancing function

code of this function.

5.2.3 Application

As an example of use, we implemented a brute-force sudoku solver. It is a funny example, but nevertheless interesting one. The international enthusiasm for the game did not spare the scientific community. There are in particular the work transforming it to a SAT problem [34] and a variety of algorithms and heuristics to reduce the size of the problem.

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 goal is to fill the entire grid so that each row, each column and each sub-grid contains one and only one each of 1 to $n^2$ numbers.

We will not use the mathematical properties of the problem, we just apply to find a solution by simple brute force to test our backtracking algorithm. From this point of view, the problem has the peculiarity of having a connection rate which varies, making the necessary rebalancing of data.

For our backtracking algorithm, we generate the children by trying every possibility for each free square and checking for validity. In this manner, the rate of turnout will be high at the start line, and low in end of line, column or sub-grid.

Fig 5.4 and Fig 5.5 show the results for a 20 dual-core cluster (Intel E2180 at 2GHz with 2Gb of memory) interconnected by a Gigabyte Ethernet network. The execution time of just searching a solution is too uncertain to have any real meaning in terms of performance – especially as the $n - 1$ nodes are explored in a different order depending on the number of processors. We have therefore chosen to modify the program to count the number of nodes explored per second, the acceleration is in relation to this number (which exceed 500 000 for 40 cores). The accelerations do not scale very well. This is mainly due to a naive balancing algorithm: too many node are exchanged at each barrier. Better algorithm could be find for specific NP-problems using realistic heuristics (and not a generic brutal force method) but this is not the subject of this article.
Figure 5.4: Speed of execution for solving a $16 \times 16$ sudoku (backtracking)

Figure 5.5: Same speed of execution versus linear acceleration
Chapter 6

Conclusion

6.1 Conclusion

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.

We present here a new syntax for our high-level BSP language: BSML (a BSP extension for ML programming language OCaml). It allows to represent easily parallel vector whose introduce parallel computations in the program and to easily used past parallel values inside new parallel computations.

This new syntax reduces size of the code. We provide some relevant example for this assertion. Code is also simpler to be read which allows finding easily bugs and cost analysis. Our new syntax could be simulated by our past BSML primitives and vice-versa making past codes compatible and allows extensions for proofs tools possible.

This new syntax were applied to design a new manner to do pattern of parallel values and to catch exceptions (in a set). This is also syntactic sugar but which considerably reduce the size of the code (and simpler its reading).

The work presented here is tightly related to the BSP model, but the new syntax is not specific to OCaml (and also parallel patterns or exceptions schemes it is based on). ML programming is classically studied for programming language extensions. Hence, we believe that there would be little work involved in translating it to, for instance, a Java or Python implementation of BSP. For example, in Java any $se$ could be a syntactic sugar to new ParArray<T>e’ with the appropriate T and e’ where ParArray are distributed arrays.

Also, we think that our patterns could applied to rewriting/patterns languages such as MGS [25] or TOM [2] which do non conventional computing by rewriting terms using patterns. As parallel vectors are structures with execution fixed size (size is always $p$ the fix number of processors), it seems possible to extends these languages by this new syntax to also easily match such structure.

6.2 Future work

Even if for matching the set of local exceptions in the trypar ... withpar, our parallel patterns are sufficient, it is not really the case for general matching of structured values. Indeed, parallel vectors can be insert in list, tuple, or any type construction in ML languages. For example is a function that transform a list of vector to a list by concating all the elements. It could be code as follow:

\[
\begin{align*}
\text{let rec to_list l accu = matchpar l withpar} \\
| [] &\rightarrow accu \\
| \text{ParArray}(x) &\rightarrow \text{to_list tl (List.concat x accu)}
\end{align*}
\]

Currently, is could be coded as:

\[
\begin{align*}
\text{let rec to_list l accu = match l with} \\
| [] &\rightarrow accu \\
| \text{hd:tl} &\rightarrow \text{matchpar hd withpar} \\
&\hspace{1cm} \text{ParArray}(x) &\rightarrow \text{to_list tl (List.concat x accu)}
\end{align*}
\]

Another example, ones can match a pair of vector:

\[
\begin{align*}
\text{matchpar x withpar} \\
| \text{ParArray}(\text{pat1}) &\rightarrow \text{ParArray}(\text{pat2}) &\rightarrow e1
\end{align*}
\]
which is not possible in a single super-step using our current pattern. It could be code as:

```ml
matchpar (fst x) withpar
| ≪ pat1’ ≫, ≪ pat2’ ≫ → e2
| _ → e3

but that used two super-steps. A more efficient solution is to transform the pair of vector to a vector of pair and match this vector:

```ml
let pair_vec_to_vec_pair v,w = ≪ v,w ≫

matchpar (pair_vec_to_vec_pair x) withpar
| ≪ pat1,pat2 ≫ → e1
| ≪ pat1’,pat2’ ≫ → e2
| ≪ _,_ ≫ → e3

but that is not as natural as a ML programmer would want. In fact, for the \( n \) patterns of vector in a parallel matching, programmers are still to have an unique super-steps and not \( n \).

More works is also needed to improve performance of the parallel pattern. Indeed, we currently multi-cast the values of the parallel vector (by doing a proj). In the case of multiple patterns, that could be improve by matching all the possible chooses, and just exchange this choose, choosing the first one that is matching per all processors and finally performs the needed exchange of values. That could be more efficient in case of small \( L \) or when data to exchange are important in size. For example, in the following parallel match:

```ml
withpar
| ≪ (Good_Found y), (Bad_Found z) * \_ \_ ≫ → y
| ≪ (Bad_Found z) * \_ \_ ≫ → raise Pb
| ≪ (Not_found) * ≫ → raise Not_found

if “z” is a big value, it is not useful to globally exchange all the values because “z” is not use in the rest of the code. A type system could improve this. Also, the BSP parameters (\( L \) and \( g \)) can allow to dynamically choose the more efficient method to perform the parallel matching. The type system for BSML of [22] would be the base for this work.
Bibliography


