# Modular Implementation of Dense Matrix Operations in a High-level BSP Language

Sovanna Tan and Frédéric Gava
*Laboratory of Algorithms, Complexity and Logic, University of Paris-East, Créteil-Paris, France,*
*sovanna.tan@univ-paris-est.fr and frederic.gava@univ-paris-est.fr*

**POSTER PAPER**

## ABSTRACT

*BSML is a high-level language for programming parallel algorithms. Built upon the OCaml language, it provides a safe setting for the implementation of BSP algorithms and for avoiding concurrency related problems (deadlocks, indeterminism,* etc.*). Dense matrices appear in many scientific computations but many libraries are limited to matrices of numeric elements. This paper is our first experiment to design a generic library of BSP implementation in ML of dense matrix operations for scientific computation.*

**KEYWORDS:** BSP, Matrix, Module, ML.

## 1. INTRODUCTION

***Generalities.*** Matrices play an important role in numerical analysis and in scientific computations in general. It is important to have efficient parallel algorithms for them. But it is not sufficient, especially for high-level language and for symbolic computations.

There exists many efficient numerical libraries (which contain matrix implementation) for low level languages C, Fortran and for higher level ones such as C++, Python, Java *etc.* We can cite BLAS[1] and ScaLAPACK[2]. Most of them only provide floating point arithmetic (in single or double precision) and sometimes complex numbers for matrix computations. Even if it suffices for most parallel computations, what about polynomials or more complex expressions? Some libraries are more generic using C++ STL templates as MTL[3] or MET[4]. But when storing non-numerical types, some mathematical functions are missing. Some might cause compiler errors or errors during runtime.

That makes the design of new and robust parallel libraries an important area of research. Creating such a library involves a trade-off between the possibility of writing predictable and efficient programs and the abstraction of such features to make programming safer and easier.

***BSML.*** An interesting compromise for coding parallel libraries is Bulk-Synchronous Parallel (BSP) ML (BSML), an extension of ML to code BSP algorithms [1]. It combines the high degree of abstraction of ML with the scalable and predictable performances of BSP.

***Main goal.*** The main objective of this research is to design generic libraries by abstracting as much as possible the details of the implementation of complex data structures such as matrices, graphs, *etc.* without generating poor performances and without writing unusable API or unreadable code. We follow the spirit of [2] by using the module language of ML and [4] by allowing to replace the primary data structures of the application by their parallel versions: therefore the complexity of parallelism is hidden behind an interface that tries to stay as close as possible to a sequential one, thus making the code easy to learn and to use.

This paper is our first experience with this design. We have chosen dense matrices because they appear in many scien-

[1] `http://www.netlib.org/blas/`
[2] `http://www.netlib.org/scalapack/`
[3] `http://www.osl.iu.edu/research/mtl/`
[4] `http://met.sourceforge.net/`

tific computations. Square dense matrices are certainly, the most studied data structures.

## 2. FUNCTIONAL BSP PROGRAMMING

For lack of space, we refer to [1][5] for a gentle introduction to the BSP model and BSP C programming (BSPlib).

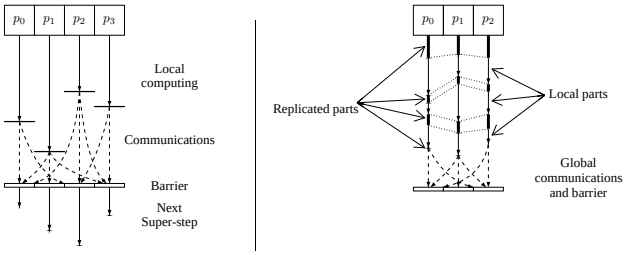


**Figure 1. A BSP Super-step (Left) and BSML Model of Execution (Right)**

***General description.*** BSML is currently 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. Other reasons include the amount of libraries available and the tools provided. We plan a full language implementation by generating 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 allows the parallelization to be done incrementally from a sequential program.

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 vector has type 'a par and embeds $\mathbf{p}$ values of type 'a at each of the $\mathbf{p}$ different processors of the BSP machine. The number of processors $\mathbf{p}$ is defined as a constant **bsp_p** throughout the execution of the program. We use the following notation to describe a parallel vector:$\langle x_0, x_1, \ldots, x_{\mathbf{p}-1} \rangle$.

***Model of Execution.*** What distinguishes this structure from an usual vector of size $\mathbf{p}$ is that the different values, that will be called *local*, are blind from each other, as it is only possible to access the local value $x_i$ in two cases: first locally, on processor $i$ (by the use of a specific primitive) and secondly, 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, the BSP costs easier to compute [3] and we think programs more readable. Worth noting is that parallel vectors can not be embedded in themselves since the BSP machine has only one level of parallelism. We refer to [6] for discussions on the problematic BSP implementation of nested vectors.

Since a BSML program deals with a whole parallel machine and individual processors at the same time, a distinction between the levels of execution will be needed (see right scheme in Fig. 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. It is used to coordinate the work of each processors.
- **Local** execution is what happens inside vectors, on each of their components: the processor uses its local data to do computations that may be different from the others. Replicated and Local execution are strictly disjoint, and typically, processors alternate between them.

The distinction between these levels is strict [6].

***Parallel primitives.*** Parallel vectors are handled through the use of different communications primitives that constitute the core of BSML. Their implementation relies either on MPI, BSPlib or on TCP/IP. The user can choose. A toplevel is also provided where the user can define its number of processors: execution on the toplevel or on a parallel machine gives the same results — except in time. Fig. 2 subsumes the use of the primitives. Informally, primitives work as follows. Let $\ll$ x $\gg$ be the vector holding x everywhere — on each processor. The $\ll \gg$ indicates that we enter a local section and pass to the local level. Replicated information is available inside the vector. Now, to access local information, we add the syntax $x$ to open the vector x and get the local value it contains, which can obviously be used only within local sections.

The **proj** primitive is the only way to extract a local value from a vector. Given a vector, it returns a function such that when it is applied to the pid of a processor, it returns the value of the vector at this processor. The **proj** primitive performs communications in order 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 the choice of functions of type (int $\rightarrow$ 'a) in **proj**. Arrays of size $\mathbf{p}$ or lists could have been chosen instead, but the interface is more functional and generic this way. Furthermore, as seen in the

[5] http://en.wikipedia.org/wiki/Bulk_Synchronous_Parallel

examples, the conversion between one style and the other is easy. Internally, our implementation relies on arrays.

The **put** primitive 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 $i$. The result of **put** is another vector of functions. Each function returns the data *received from* processor $i$ when applied to $i$.

BSP paradigm's simplicity and elegance comes at a cost: the ability to synchronize a subset of the processors would break the BSP cost model. Subset synchronization is generally used to recursively decompose computations into independent tasks — the divide-and-conquer paradigm. The last primitive allows the evaluation of two BSML expressions $E_1$ and $E_2$ as super-threads. From the programmer's point of view, the semantics of the **super** is the same as pairing *i.e.*, building the pair $(E_1, E_2)$ but the evaluation of **super** $E_1$ $E_2$ is less costly because it merges the communication and the synchronization phases of $E_1$ and $E_2$.

Defining some useful libraries simplify the coding of the algorithms. Some typical examples of BSML programming can be found in [4, 3, 7, 6].

## 3. A MODULAR LIBRARY

### 3.1. The OCaml's module system

In this section we briefly recall the characteristics of the OCaml's module system. This is an independent high-order language (strongly typed [9]) above OCaml, completing only software engineering functionalities: separate compilation, structuring space names and genericity of the code. The building blocks are structures which pack together related definitions. Here is for instance a structure packaging together a type of matrix and some operations:

```
module Matrices =
 struct
   type 'a mat = 'a array array
   (* plus: ('a→ 'a→ 'a)→ mat→ mat→ mat *)
   let plus oplus m1 m2 = ...
 end
```

where "oplus" is an addition operator for elements of the matrix. Outside the structure, its components can be referred by using the structure name, for instance, "Matrices.plus" or "Matrices.mat" in a typed context. Signatures are interfaces for structures. A signature specifies which components are accessible from the outside, and with which type. For instance, the signature below specifies our matrices by hiding the implementation:

```
module type TypeMatrices =
  sig
    type 'a mat
    val plus : ('a → 'a → 'a) → 'a mat → 'a mat → 'a mat
  end

module OurMatrices = (Matrices : TypeMatrices)

(* or we can directly write *)
module OurMatrices: TypeMatrices =
 struct
  type 'a mat = 'a array array
  ...
 end
```

Functors are "functions" from structures to structures. They are used to express parametrized structures. For instance, here is a structure implementing matrices parametrized by a structure providing the type of the elements of the matrices and operators on them:

```
module type Algebra =
 sig
   type t
   val plus: t → t → t
 end

module type TypeMatrices =
  sig
    type element
    include Algebra
  end

module Matrices (Elt: Algebra) : TypeMatrices with type element = Elt.t =
 struct
   type element = Elt.t
   type t= element array array
   let plus m1 m2 = ...
 end
```

Note that "Algebra" is a sub-type of "TypeMatrices" and its thus allows to define a module of matrices of matrices:

```
module MatofMat=Matrices(Matrices(struct type t=float let plus=(+.) end))
```

As functors are first-class values in the module language (high-order language), they can also be used as arguments to another functor (it is the same kind of abstraction than functions as arguments of other functions). For instance, we can build a parallel implementation of matrix as a parallel vector of matrices but where the sequential implementation of them is abstract by a function (the parallel implementation is independent of the sequential one):

```
module Make (Elt : Algebra) (MakeLocMat:functor(Elt:Algebra) →
TypeMatrices with type element=Elt.t):TypeMatrices with type element=Elt.t
 = struct
  module LocMat = MakeLocMat(Elt)
  type element = Elt.t
  type t = LocMat.t par
  let plus = ...
 end
```

### 3.2. The BSML linear algebra Module

In this section, we describe the structure of our linear algebra entities[6]. We begin by defining an hierarchy of module types also called interfaces for the algebraic structures

[6] Freely available at `http://lacl.univ-paris12.fr/gava/bsmlLA/`

| primitive | type | informal description |
|---|---|---|
| $\ll$ x $\gg$ | $t$ par (if $x : t$) | $\langle x, \ldots, x \rangle$ |
| $pid$ (within a vector) | int | value $i$ on processor $i$ |
| $v$ (within a vector) | $t$ (if $v : t$ par) | $v_i$ on processor $i$ (if $v = \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$) |
| proj | 'apar→ (int→ 'a) | $\langle x_0, \ldots, x_{p-1} \rangle \mapsto (\mathsf{fun}\ i \rightarrow x_i)$ |
| put | (int→ 'a)par→ (int→ 'a)par | $\langle f_0, \ldots, f_{\mathbf{p}-1} \rangle \mapsto \langle (\mathsf{fun}\ i \rightarrow f_i\ 0), \ldots, (\mathsf{fun}\ i \rightarrow f_i\ (\mathbf{p}-1)) \rangle$ |
| super | (unit→ 'a)→ (unit→ 'a)→ 'a∗'b | $f_a \mapsto f_b \mapsto (f_a\ (), f_b\ ())$ |

**Figure 2. Summary of BSML Primitives**

which are involved in linear algebra. We start from Set, and move towards Group, Ring, Field and Normed Field. In each interface, we declare the type of each structure operator and the name of neutral elements. For normed structures, we add the type of the norm and its comparison operators. These modules were inspired by an existing library for formal and numerical calculus in OCaml[7].

We also introduce other functions such as absolute value, modulus, square root and a very small value called epsilon needed by some classical linear algorithms such as LU decomposition or Gauss Jordan elimination. With these strong typing constraints, our algorithms work with any data type that implements the required interface. This way we avoid many compiler or runtime errors and build a robust library. To do actual computations, we use existing OCaml data types to define the fields of real and complex numbers according to the specified normed field interface.

In this framework, a matrix is implemented with a BSML parallel vector of normed field element arrays. Linear algebra algorithms are functors which perform computation on matrices made of normed field elements.

For matrix computation, it is convenient to denote the processors by a two dimensional array $\mathbf{P_r} \times \mathbf{P_c}$. Parallel algorithms generally distribute matrix entries to processors in such a way that a matrix row is located on a processor row and a matrix column on a processor column. Many BSP algorithms have been analyzed with a scattered distribution [1], [5] where matrix element $(i, j), 0 \leq i \leq m, 0 \leq j \leq n$ is located on processor $(i \mod \mathbf{P_r}, j \mod \mathbf{P_c})$. Therefore we choose first to implement these algorithms with a scattered distribution.

## 4. DENSE MATRIX OPERATIONS

### 4.1. Dense Matrix Multiplication

This first example is based on an algorithm presented in [5] which is independent of type of data of the square matrices. The true algorithm implemented in our library is close to it except that it works for arbitrary size matrices and used a scattered distribution and not a block one (it is thus less readable). Initially, two $n \times n$ matrices $A$ and $B$ are distributed among the $\mathbf{p} = \sqrt{\mathbf{p}} \times \sqrt{\mathbf{p}}$ processors so that each processor stores a sub-set (call block) of size $m \times m$ (where $m = \frac{n}{\sqrt{\mathbf{p}}}$) 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 $(\frac{j}{m}) \times \sqrt{\mathbf{p}} + \frac{i}{m}$-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:

**begin** Mult(C,A,B)
  **let** $m = \frac{n}{\sqrt{\mathbf{p}}}$
  **let** $p_i = pid \mod \sqrt{\mathbf{p}}$ **and** $p_j = \frac{pid}{\sqrt{\mathbf{p}}}$ **and** $C_q = [0]$ **in**
  **for** $0 \leq l < \sqrt{\mathbf{p}}$ **do**
    **begin**
      **let** $a = A_{((p_i+p_j+l) \mod \sqrt{\mathbf{p}}) \times \sqrt{\mathbf{p}} + p_i}$
      **and** $b = B_{((p_i+p_j+l) \mod \sqrt{\mathbf{p}}) + p_j \times \sqrt{\mathbf{p}}}$ **in**
      $C_{pid} \leftarrow C_{pid} \oplus a \otimes b$
    **end**
**end** Mult

One can remark that each processor receives data from two distinct processors at each super-step due to a *round-robin* distribution of the blocks. In Fig 3, we give the code of this algorithm using BSML and C+BSLlib [1]. For the BSML implementation, we use twice the "get_from" utility function and we "superposed" them to avoid the duplication of super-steps. Each "get_from" is built as in the algorithm and as the bsp_get in the C code. In this C code, we used specific sequential addition and multiplication of matrices of floats. These two codes have two main differences. First, using our modular implementation, the BSML code abstracts the type of the elements of the matrices. Second, in the C code we modify in place the matrix while in the BSML code, the resulting matrix is build dynamically.

All the tests were run on the LACL cluster composed of 20 Pentium dual core 2Ghz with 2GB of RAM interconnected with a Gigabyte Ethernet network. Fig. 4 gives some benchmarks for matrices of floats (times and speedups). Note that if we want other kinds of matrices, using the

[7] `http://www.lama.univ-savoie.fr/~RAFFALLI/formel.html`

```
(* multiply_par:t→ t→ t *)
let multiply_par parA parB =
 let parC=ref mat_create neutral parA.size_row in
 let sqrt_p=sqrt_int bsp_p in
 let ni=n/sqrt_p in
  let pi=≪ $pid$ mod sqrt_p≫
 and pj=≪ $pid$/sqrt_p≫ in
    let fromA=≪ fun l→ (($pi$+$pj$+l) mod sqrt_p)∗sqrt_p+$pi$≫
  and fromB=≪ fun l→ (($pi$+$pj$+l) mod sqrt_p)+$pj$∗sqrt_p≫
 in
  for l=0 to sqrt_p−1 do
   let rcvpA,rcvpB = super (fun ()→ get_from fromA parA l)
                           (fun ()→ get_from fromB parB l)
    in ≪ $parC$ := seq_plus $parC$ (seq_mult $rcvpA$ $rcvpB$) ≫
  done;
  parC
```

```
void c_mat_mul(double *A,
               double *B,
               double *C,
               int n,int p_sqrt)
{
 register int pid;
 double *a,*b,*c;
 register int l,pi,pj,ni,size
 ni=n/p_sqrt;
 size=ni*ni*sizeof(double);
 pi=pid % p_sqrt;
 pj=pid / p_sqrt;
 a = (double *)malloc(size);
 b = (double *)malloc(size);
 c = (double *)malloc(size);
 init_mat_float(C,ni)
 bsp_push(A,size);
 bsp_push(B,size);
 bsp_sync();
 for (l=0;l<p_sqrt;++l)
  {
   bsp_get((((pi+pj+l)%p_sqrt)*p_sqrt+pi),A,0,a,size);
   bsp_get((((pi+pj+l)%p_sqrt)+pj*p_sqrt),B,0,b,size);
   bsp_sync();
   mat_seq_mult_float(a,b,c)
   mat_seq_add_float(C,c)
  }
 bsp_pop(B);
 bsp_pop(A);
 free((void *)b);
 free((void *)a);
 free((void *)c);
}
```

**Figure 3. BSML Code of the Matrix Multiplication (Left) and in C+BSPlib (Right, Two Columns)**

modular BSML code, we just have to change the instantiation of the algebra while using C code, we have to change the code itself. We can remark that the C+BSPlib code is three time faster than BSML one. This mainly due to the use of functors that are known by the OCaml community to slow down the programs. For both, speedup are close to the linear acceleration which is not surprising since matrix multiplication is a good parallel problem. We can also see a super-linear acceleration for the BSML code which is more scalable than the C code. This is mainly due to the fact that the computational parts of the C code are efficient enough (even in the sequential implementation) to be only slow down by the communications.

### 4.2. LU decomposition

In linear algebra, LU decomposition is the decomposition of an $n \times n$ dense matrix $A = a_{ij}$ into a a lower triangular matrix $L$ and an upper triangular matrix $U$ so that $A = L \times U$. This method has a great deal of applications in numerical analysis (to solve systems of linear equations or calculate the determinant). For lack of space, we do not show the BSML code nor the C one which comes from [1]. Fig 5 presents the benchmark that we have done on our cluster with matrices of floats.

We can see that performances degrade beyond 10 processors for both the ML and C codes. We can also see a super-linear acceleration for the C code (this is certainly due to less missed caches) with less than five processors. It can be seen that the larger the matrix is, the less degradation appears quickly and is important. This is due to the fact that the algorithm communicates a lot and that the report computation times/global synchronization is no more to the advantage of a large number of processors. Each processor has enough data to process: for the LU decomposition of matrices of complex numbers, this degradation occurs only for a number of processors twice as large.

As in the multiplication, the ML code is three times slower. This suggests that it is not useless to continue the work on parallel extensions to the ML languages but we must consider that these languages offer safety and are more efficient in the context of non-floating calculations [8]. Moreover, this safety enforcement could be an asset when parallel computations performed are critical.

We have also implemented a Cholesky factorization, a Gauss Jordan elimination, transposition and inversion of matrices. Many other operations can be easily implemented from this set of operations. For lack of space, we do not presented them here.

## 5. RELATED WORKS

[8] is the first book that gives some useful examples of the use of OCaml for scientific computing. It shows the advantage of a polymorphic, safe, efficient language for this community. Matrices are treated as arrays of arrays of floats. But there is no study of generic implementations. [4] presents the first implementation of modular BSP data structures in ML: parallel set, map, hash tables, lifo are provided with an application to scientific computation. The first massive use of functors for a data structure library is OCamlGraph [2]. It provides a generic and modular implementation of graph in OCaml without poor performances. BSML extension of this work is a work in progress.

For OCaml, we have found three libraries for manipulating matrices: psilab[8] and lacaml[9] which both used LAPACK for providing matrices operations. Psilab has very convenient printing capacities for matrices. The last one, OCamlFloat[10] is also an interface to the Lapack and Blas libraries which aims to improve the clarity and efficiency

[8] http://psilab.sourceforge.net/
[9] http://hg.ocaml.info/release/lacaml
[10] http://www.mirrorsky.com/ocaml/

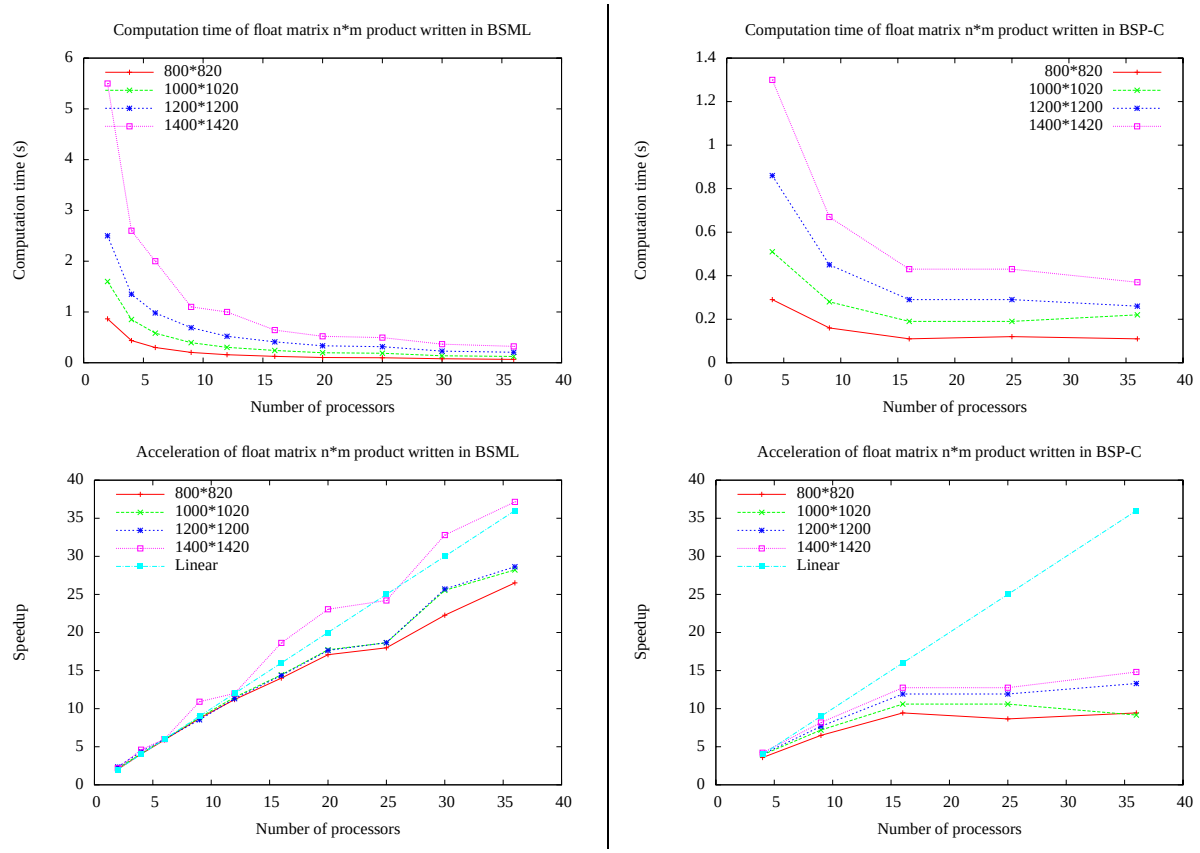


**Figure 4. Multiplications of Dense Matrices of Floats**

of numerical algorithms. This library also allows to build a matrix using sub-matrices and there is a common interface for matrices of floats and complex numbers. But there is no interface for other kind of data. Using C++ template, we can found many generic implementations of matrices[11] in the literature and on the web. Some of them use parallel algorithms[12]. To our knowledge, none of them offers all together parallel operations, true generic interface and safety/portability of execution. The matrix library scipy[13] for the Python language also lacks these properties.

## 6. CONCLUSION

In this paper, we presented a generic, modular and ML style implementation of some well known linear algebra problems in the context of the BSP model, analysed their performance and compared them to a standard C implementation. We reported the experimental results obtained:

- The experiments have strengthened our convictions that algorithm design based on the BSP model is plausible, easy to follow, leads to algorithms that can be portable, scalable and without much programming effort, quite efficient;

- It seems that the pure numerical algorithms used as examples in the paper are not the more appropriate to show the advantages of this programming environment. Using the generic possibilities of the ML languages, parallel programs can be written to work on any kind of algebra and not only numerical ones. We think about many biological/chemistry systems where elements are not floats but more complex structures (*e.g.* atoms, cells, *etc.*);

- This genericity clearly slows down programs but not as much as one might think. This is a well known fact in the C++/template community.

The algorithms presented and their implementations are not the better ones especially in the case of more specific dense matrices ( such as Vandermonde, Cauchy matrices, complex Hadamard matrices, *etc.*). Future work will consider three important points.

[11] *e.g.* `http://www.codeproject.com/KB/architecture/ymatrix.aspx`

[12] `http://polaris.cs.uiuc.edu/hta/`

[13] `http://www.scipy.org/`

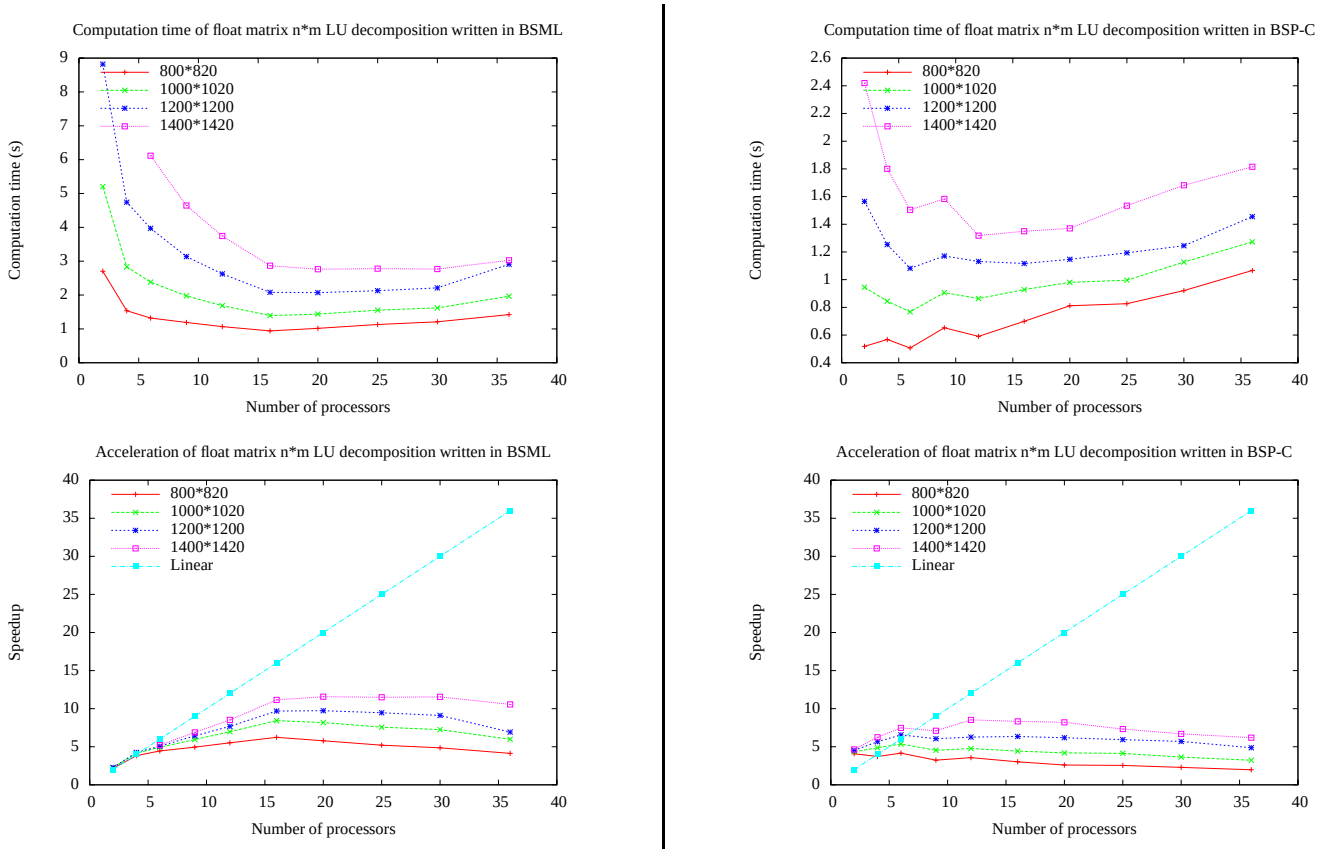


**Figure 5. LU Decomposition of Dense Matrices of Floats**

First we introduce squares matrices and conversion to/from dense ones and naturally generic implementation of the specific square matrix algorithms. Then, we can also consider the big zoo of kinds of matrices and imagine specific functor for each of them (implemented algorithms can be the same but types would be different).

Secondly, we are thinking to compare the performance of our work with that of functional language based related works. To better judge the readability of the proposed codes, we need to provide quantitative comparisons — sizes of the codes, timings, *etc*. It is a hard work since comparing programming languages is generally difficult.

In the third place, matrices are often very close to graphs. There exists a generic implementation of graphs in OCaml [2]. Implementing a BSP version of this library and merging it with our library is a challenging future work.

## REFERENCES


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

[2] S. Conchon, J.-C. Filliâtre, and J. Signoles. "Designing a Generic Graph Library using ML Functors". In *Trends in Functional Programming*, volume 8. Intellect, 2007.

[3] F. Gava. "BSP Functional Programming; Examples of a cost based methodology". In *ICCS*, volume 5101 of *LNCS*, pages 375–385. Springer, 2008.

[4] F. Gava. "A modular implementation of data structures in bulk-synchronous parallel ML". *Parallel Processing Letters*, 18(1):39–53, 2008.

[5] A. V. Gerbessiotis. "Algorithmic and practical considerations for dense matrix computations on the BSP model". Technical Report PRG-TR-32-97, The University of Oxford, 1997.

[6] L. Gesbert. *"Développement systématique et sureté d'éxécution en programmation paralléle structurée"*. PhD thesis, University of Paris-East, 2009.

[7] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski. "Bulk Synchronous Parallel ML with Exceptions". *Future Generation Computer Systems*, 2009. to appear.

[8] J. D. Harrop. "Objective Caml for Scientists", 2005.

[9] X. Leroy. "A modular module system". *Journal of Functional Programming*, 10(3):269–303, 2000.