A Parallel Algorithm for solving BSDEs

Céline Labart, Jérôme Lelong

A Parallel Algorithm for solving BSDEs

Céline Labart[1,3] Jérôme Lelong[2,3]

March 19, 2012

Abstract

We present a parallel algorithm for solving backward stochastic differential equations. We improve the algorithm proposed in Gobet and Labart (2010), based on an adaptive Monte Carlo method with Picard's iterations, and propose a parallel version of it. We test our algorithm on linear and non linear drivers up to dimension 8 on a cluster of 312 CPUs. We obtained very encouraging speedups greater than 0.7.

Keywords : backward stochastic differential equations, parallel computing, high performance computing, Monte-Carlo methods.

1 Introduction

Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a given probability space endowed with a d-dimensional standard Brownian motion W, whose natural filtration, augmented with $\mathbb{P}$-null sets, is denoted $(\mathcal{F}_t)_{0\leq t\leq T}$ (T is a fixed terminal time). We denote by $(Y^{t,x}, Z^{t,x})$ the solution of the following backward stochastic differential equation (BSDE)

$$-dY_s^{t,x} = f(s, X_s^{t,x}, Y_s^{t,x}, Z_s^{t,x})ds - Z_s^{t,x}dW_s, \;\; Y_T^{t,x} = \Phi(X_T^{t,x}), \;\; t \leq s \leq T \tag{1.1}$$

where $f : [t,T] \times \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}$, $\Phi : \mathbb{R}^d \to \mathbb{R}$ and $X^{t,x}$ is the $\mathbb{R}^d$-valued process solution of

$$X_s^{t,x} = x + \int_t^s b(r, X_r^{t,x})dr + \int_t^s \sigma(r, X_r^{t,x})dW_r, \;\; t \leq s \leq T \tag{1.2}$$

with $b : [t,T] \times \mathbb{R}^d \to \mathbb{R}^d$ and $\sigma : [t,T] \times \mathbb{R}^d \to \mathbb{R}^{d\times d}$. The process Y is real valued, whereas the process Z has values in $\mathbb{R}^d$ (as a row vector).

Solving BSDEs numerically is a very challenging issue. On a single processor system, it can require several hours of computation for high values of d. Recent advances in parallel computing hardwares such as multi–core processors, clusters and GPUs are then of high interest. Several algorithms to solve BSDEs can be found in the literature. Ma et al. (1994) presented an algorithm to solve quasilinear PDEs (associated to forward BSDEs) using a finite difference approximation. Concerning algorithms based on the dynamic programming equation, we refer to Bouchard and Touzi (2004), Gobet et al. (2005), Bally and Pagès

[1] Laboratoire de Mathématiques, CNRS UMR 5127, Université de Savoie, Campus Scientifique, 73376 Le Bourget du Lac, France. labart@univ-savoie.fr.
[2] Laboratoire Jean Kuntzmann, Université de Grenoble and CNRS, BP 53, 38041 Grenoble, Cedex 09, France jerome.lelong@imag.fr.
[3] Projet MathRisk, INRIA Paris–Rocquencourt.

(2003) and Delarue and Menozzi (2006). In Bouchard and Touzi (2004), the authors compute the conditional expectations appearing in the dynamic programming equation using Malliavin calculus techniques, whereas Gobet et al. (2005) propose a scheme based on iterative regression functions, approximated by projections on a reduced set of functions — the coefficients of the projection being evaluated using Monte Carlo simulations. Bally and Pagès (2003) and Delarue and Menozzi (2006) use quantization techniques for solving reflected BSDEs and forward BSDEs respectively. Bender and Denk (2007) propose a forward scheme which avoids the use of nested conditional expectations backward in time. Instead, it mimics Picard's type iterations for BSDEs and, consequently, has nested conditional expectations along the iterations. Our approach is based on the algorithm developed by Gobet and Labart (2010) which combines Picard's iterations and an adaptive control variate to solve the associated non linear PDE (called PACV algorithm in the following). Compared to the algorithms based on the dynamic programming equation, the PACV algorithm provides regular solutions in time and space (which is coherent with the regularity of both the option price and its delta).

In this paper, we propose a parallel version of the algorithm developed by Gobet and Labart (2010), after having replaced the kernel operator by an extrapolating operator to approximate functions and their derivatives. The parallelization is far from being as simple as for crude Monte-Carlo algorithms. We explain the difficulties encountered when parallelizing it and how we solved them. Our implementation relying on MPI for passing messages is therefore suitable for clusters. Finally, we present and discuss the performances of our approach for solving BSDEs appearing in financial problems, such that pricing European options in the Black–Scholes framework with a borrow rate different from the bond rate. In this context, the driver is non linear and pricing cannot be achieved using a standard Monte–Carlo approach.

The paper is organized as follows. In section 2, we briefly recall the link between BSDEs and PDEs which is the keystone of the PACV algorithm. In section 3, we describe the algorithm and discuss the choice of the approximating operator and in Section 4 we explain how the parallelization has been carried out. Finally, in Section 5, we conclude the paper by some numerical tests of our parallel algorithm for pricing and hedging European basket options under constrained portfolios in dimension up to 8.

1.1 Definitions and Notations

- Let $C_b^{k,l}$ be the set of continuously differentiable functions $\phi : (t,x) \in [0,T] \times \mathbb{R}^d$ with continuous and uniformly bounded derivatives w.r.t. t (resp. w.r.t. x) up to order k (resp. up to order l).
- C_p^k denotes the set of C^{k-1} functions with piecewise continuous k^{th} order derivative.
- For $\alpha \in]0,1]$, $C^{k+\alpha}$ is the set of C^k functions whose k^{th} order derivative is Hölder continuous with order α.

2 Link between BSDEs and non linear PDEs

From now on, we assume the following Hypothesis, which ensures among others existence and uniqueness of solutions to Equations (1.1)-(1.2).

Hypothesis 1

- *The driver f is a bounded Lipschitz continuous function, i.e, for all (t_1, x_1, y_1, z_1), $(t_2, x_2, y_2, z_2) \in [0,T] \times \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^d$, $\exists\, L_f > 0$,*

$$|f(t_1, x_1, y_1, z_1) - f(t_2, x_2, y_2, z_2)| \leq L_f(|t_1 - t_2| + |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|).$$

- *σ is uniformly elliptic on $[0,T] \times \mathbb{R}^d$, i.e, there exist two positive constants σ_0, σ_1 s.t. for any $\xi \in \mathbb{R}^d$ and any $(t,x) \in [0,T] \times \mathbb{R}^d$*

$$\sigma_0 |\xi|^2 \leq \sum_{i,j=1}^{d} \xi_i [\sigma\sigma^*]_{i,j}(t,x)\xi_j \leq \sigma_1 |\xi|^2.$$

- *Φ is bounded in $C^{2+\alpha}$, $\alpha \in]0,1]$.*
- *b and σ are in $C_b^{1,3}$ and $\partial_t \sigma$ is in $C_b^{0,1}$.*

Let us also recall the link between BSDEs and semilinear PDEs, which is the keystone of the PACV algorithm. We refer to Pardoux and Peng (1992) or El Karoui et al. (1997) for a proof of the following result.

We can link the solution (Y, Z) of the BSDE (1.1) to the solution of a PDE.
Let u be a $C^{1,2}$ function satisfying $|u(t,x)| + |\partial_x u(t,x)\sigma(t,x)| \leq C(1 + |x|)$ and solving

$$\begin{cases} \partial_t u(t,x) + \mathcal{L}u(t,x) + f(t,x,u(t,x),(\partial_x u\sigma)(t,x)) = 0, \\ u(T,x) = \Phi(x), \end{cases} \tag{2.1}$$

with $\mathcal{L}$ defined by

$$\mathcal{L}_{(t,x)} u(t,x) = \frac{1}{2} \sum_{i,j} [\sigma\sigma^*]_{ij}(t,x) \partial^2_{x_i x_j} u(t,x) + \sum_i b_i(t,x) \partial_{x_i} u(t,x).$$

Then, $u(t,x) = Y_t^{t,x}$ and

$$\forall s \in [t,T], \quad (Y_s^{t,x}, Z_s^{t,x}) = (u(s, X_s^{t,x}), \partial_x u(s, X_s^{t,x})\sigma(s, X_s^{t,x})). \tag{2.2}$$

It remains to know how to ensure that PDE (2.1) has a unique solution.

Theorem 1 (Delarue and Menozzi (2006), Theorem 2.1)**.** *Under Hypothesis 1, PDE (2.1) admits a unique solution which belongs to $C_b^{1,2}$.*

3 Presentation of the PACV Algorithm

3.1 Description

We present the algorithm introduced by Gobet and Labart (2010) to solve standard BSDEs. Assume that we want to solve BSDE (1.1) with X starting from x at time 0. In the following, for the sake of clearness, we may omit the upper index $(0,x)$ when there is no possible

confusion and use the notations X (resp. (Y,Z)) instead of $X^{0,x}$ (resp. $(Y^{0,x},Z^{0,x})$). Then, (2.2) gives

$$\forall t\in[0,T],\ \ (Y_t,Z_t)=(u(t,X_t),\partial_x u(t,X_t)\sigma(t,X_t)).$$

Then, solving BSDE (1.1) is equivalent to solving the semilinear PDE (2.1) on $[0,T]$. The current algorithm provides a converging sequence of approximations $(u^k)_k$ of the solution of this PDE, based on Picard's iterations combined with an adaptive Monte–Carlo method. We need to approximate u on the whole domain $[0,T]\times\mathbb{R}^d$ (since we want to get $(Y_t,Z_t)_{0\leq t\leq T}$). Then, we need to compute each approximation u^k on a grid of points.
Let u^k denote the approximation of the solution u of (2.1) at step k. If we are able to compute an explicit solution of (1.2), the approximation of (Y,Z) at step k follows from (2.2): $(Y_t^k,Z_t^k)=(u^k(t,X_t),\partial_x u^k(t,X_t)\sigma(t,X_t))$, for all $t\in[0,T]$. Otherwise, we introduce the approximation X^N of X obtained with a N–time step Euler scheme:

$$\forall s\in[0,T],\ \ dX_s^N=b(\varphi^N(s),X^N_{\varphi^N(s)})ds+\sigma(\varphi^N(s),X^N_{\varphi^N(s)})dW_s, \tag{3.1}$$

where $\varphi^N(s)=\sup\{t_j\ :\ t_j\leq s\}$ is the largest discretization time not greater than s and $\{0=t_0<t_1<\cdots<t_N=T\}$ is a regular subdivision of the interval $[0,T]$. Then, we write

$$(Y_t^k,Z_t^k)=(u^k(t,X_t^N),\partial_x u^k(t,X_t^N)\sigma(t,X_t^N)),\ \text{for all } t\in[0,T].$$

The basic idea of each iteration step is the following:

$$u^{k+1}=u^k+\ \text{Monte–Carlo evaluations of the error}(u-u^k).$$

By combining Itô's formula applied to $u(s,X_s^{t,x})$ and $u^k(s,X_s^{N,t,x})$ ($X^{N,t,x}$ denotes the approximation of $X^{t,x}$ using an Euler scheme with N time steps) between t and T and the semilinear PDE (2.1) satisfied by u, we get that the correction term is given by

$$(u-u^k)(t,x)=\mathbb{E}\left[\Psi\left(t,x,f_u,\Phi,W\right)-\Psi^N\left(t,x,-(\partial_t+\mathcal{L}^N)u^k,u^k(T,.),W\right)|\mathcal{G}^k\right] \tag{3.2}$$

where

- $\mathcal{G}^k$ is the σ-algebra generated by the set of all random variables used to build u^k. In other words, the conditional expectation in Equation (3.2) can be seen as a standard expectation computed only w.r.t. to the law of W, all other variables being constant.
- $\mathcal{L}^N$ denotes the generator associated to the Euler Scheme X^N, i.e.

  $$\mathcal{L}^N u(s,X_s^{N,t,x})=$$
  $$\frac{1}{2}\sum_{i,j}[\sigma\sigma^*]_{ij}(\varphi(s),X^{N,t,x}_{\varphi(s)})\partial^2_{x_ix_j}u(s,X_s^{N,t,x})+\sum_i b_i(\varphi(s),X^{N,t,x}_{\varphi(s)})\partial_{x_i}u(s,X_s^{N,t,x}),$$

- $f_v:[0,T]\times\mathbb{R}^d\to\mathbb{R}$ denotes the following function

  $$f_v:t,x\longmapsto f(t,x,v(t,x),(\partial_x v\sigma)(t,x)),$$

  where f is the driver of BSDE (1.1), σ is the diffusion coefficient of the SDE satisfied by X and $v:[0,T]\times\mathbb{R}^d\to\mathbb{R}$ is C^1 w.r.t. to its second argument.

- Ψ and Ψ^N denote

$$\Psi(t,x,g_1,g_2,W) = \int_t^T g_1(r, X_r^{t,x}(W))dr + g_2(X_T^{t,x}(W)),$$
$$\Psi^N(t,x,g_1,g_2,W) = \int_t^T g_1(r, X_r^{N,t,x}(W))dr + g_2(X_T^{N,t,x}(W)),$$

  where W denotes the standard Brownian motion appearing in (1.2) and used to simulate X^N, as given in (3.1).

Note that Ψ and Ψ^N can actually be written as expectations by introducing a random variable U uniformly distributed on $[0,1]$.

$$\Psi(t,x,g_1,g_2,W) = \mathbb{E}\left[(T-t)g_1(t+(T-t)U, X^{t,x}_{t+(T-t)U}(W)) + g_2(X_T^{t,x}(W))|W\right],$$
$$\Psi^N(t,x,g_1,g_2,W) = \mathbb{E}\left[(T-t)g_1(t+(T-t)U, X^{N,t,x}_{t+(T-t)U}(W)) + g_2(X_T^{N,t,x}(W))|W\right].$$

In the following, let $\psi^N(t,x,g_1,g_2,W,U)$ denote

$$\psi^N(t,x,g_1,g_2,W,U) = (T-t)g_1(t+(T-t)U, X^{N,t,x}_{t+(T-t)U}(W)) + g_2(X_T^{N,t,x}(W)) \tag{3.3}$$

such that $\Psi^N(t,x,g_1,g_2,W) = \mathbb{E}[\psi^N(t,x,g_1,g_2,W,U)|W]$.

From a practical point of view, the PDE (2.1) is solved on $[0,T]\times\mathcal{D}$ where $\mathcal{D}\subset\mathbb{R}^d$ is chosen such that $\mathbb{P}(\forall t\in[0,T], X_t\in\mathcal{D})$ is very close to 1.

Algorithm 1 *We begin with $u^0\equiv 0$. Assume that an approximated solution u^k of class $C^{1,2}$ is built at step k. Here are the different steps to compute u^{k+1}.*

- *Pick at random n points $(t_i^k, x_i^k)_{1\le i\le n}$ uniformly distributed over $[0,T]\times\mathcal{D}$.*
- *Evaluate the Monte–Carlo correction c^k at step k at the points $(t_i^k,x_i^k)_{1\le i\le n}$ using M independent simulations*

$$c^k(t_i^k,x_i^k) = \frac{1}{M}\sum_{m=1}^{M}\left[\psi^N\left(t_i^k, x_i^k, f_{u_k} + (\partial_t + \mathcal{L}^N)u_k, \Phi - u^k, W^{m,k,i}, U^{m,k,i}\right)\right].$$

- *Compute the vector $(u^k(t_i^k,x_i^k))_{1\le i\le n}$ and deduce the vector $(u^k+c^k)(t_i^k,x_i^k))_{1\le i\le n}$.*
- *From these values, extrapolate the function $u^{k+1}=u^k+c^k$ on $[0,T]\times\mathcal{D}$*

$$u^{k+1}(t,x) = \mathcal{P}^k(u^k+c^k)(t,x), \quad \text{for } (t,x)\in[0,T]\times\mathcal{D}, \tag{3.4}$$

  where P^k is a deterministic operator, which only uses the values of the function at the points $(t_i^k,x_i^k)_{1\le i\le n}$ to approximate the function on the whole domain $[0,T]\times\mathcal{D}$. The choice of the operator P^k is discussed in Section 3.2.

3.2 Choice of the operator

The most delicate part of the PACV algorithm is how to extrapolate a function h and its derivatives when only knowing its values at n points $(t_i,x_i)_{i=1,\dots,n}\subset[0,T]\times\mathcal{D}$. We recall that $\mathcal{D}$ is a subset of $\mathbb{R}^d$, hence each x_i is $\mathbb{R}^d$-valued.

3.2.1 A kernel operator

In the originally published PACV Algorithm, a function h was extrapolated from the values computed on the grid by using a kernel operator of the form

$$h(t,x) = \sum_{i=1}^{n} h(t_i,x_i) K_t(t-t_i) K_x(x-x_i),$$

where K_t is a one dimensional kernel whereas K_x is a product of d one dimensional kernels. Hence, evaluating the function h at a given point (t,x) required $O(n \times d)$ computations.

The convergence result established by Gobet and Labart (2010, Theorem 5.1) was based on the properties of the operator presented in Gobet and Labart (2010, Section 4). Using the linearity and the boundedness of the operator, they managed to prove that the errors $\|v - \mathcal{P}^k v\|$ and $\|\partial_x v - \partial_x(\mathcal{P}^k v)\|$ are bounded, which is a key step in proving the convergence of the algorithm. At the end of their paper, they present an operator based on kernel estimators satisfying the assumptions required to prove the convergence of the algorithm.

3.2.2 An extrapolating operator

The numerical properties of kernel operators are very sensitive to the choice of their window parameters which are quite hard to tune for each new problem. Hence, we have tried to use an other solution. Basically, we have used a least–square approach which consists in extrapolating a function by solving a least square problem defined by the projection of the original function on a countable set of functions. Assume we know the values $(y_i)_{i=1,\dots,n}$ of a function h at the points $(t_i,x_i)_{i=1,\dots,n}$, the function h can be extrapolated by computing

$$\boldsymbol{\alpha} = \arg\min_{\alpha \in \mathbb{R}^p} \sum_{i=1}^{n} \left| y_i - \sum_{l=1}^{p} \alpha_l B_l(t_i,x_i) \right|^2, \tag{3.5}$$

where $(B_l)_{l=1,\dots,p}$ are some real valued functions defined on $[0,T]\times\mathcal{D}$. Once $\boldsymbol{\alpha}$ is computed, we set $\hat{h}(t,x) = \sum_{l=1}^{p} \boldsymbol{\alpha}_l B_l(t,x)$. For the implementation, we have chosen the $(B_l)_{l=1,\dots,p}$ as a free family of multivariate polynomials. For such a choice, $\hat{h}$ is known to converge uniformly to h when p goes to infinity if $\mathcal{D}$ is a compact set and h is continuous on $[0,T]\times\mathcal{D}$. Our algorithm also requires to compute the first and second derivatives of h which are approximated by the first and second derivatives of $\hat{h}$. Although the idea of approximating the derivatives of a function by the derivatives of its approximation is not theoretically well justified, it is proved to be very efficient in practice. We refer to Wang and Caflish (2010) for an application of this principle to the computations of the Greeks for American options. In the following, we denote by PACVLS this new algorithm with the extrapolating step performed by a least-squared method.

Practical computation of the vector $\boldsymbol{\alpha}$ In this part, we use the notation $d' = d+1$. It is quite easy to see from Equation (3.5) that $\boldsymbol{\alpha}$ is the solution of a linear system. The value $\boldsymbol{\alpha}$ is a critical point of the criteria to be minimized in Equation (3.5) and the vector $\boldsymbol{\alpha}$ solves

$$\sum_{l=1}^{p} \boldsymbol{\alpha}_l \sum_{i=1}^{n} B_l(t_i,x_i) B_j(t_i,x_i) = \sum_{i=1}^{n} y_i B_j(t_i,x_i) \quad \text{for } j = 1,\dots,p, \text{ i.e.}$$

$$\boldsymbol{A}\boldsymbol{\alpha} = \sum_{i=1}^{n} y_i B(t_i,x_i) \tag{3.6}$$

where the $p \times p$ matrix $\boldsymbol{A} = (\sum_{i=1}^{n} B_l(t_i, x_i) B_j(t_i, x_i))_{l,j=1,\dots,p}$ and the vector $B = (B_1, \dots, B_p)^*$. The matrix $\boldsymbol{A}$ is symmetric and positive definite but often ill-conditioned, so we cannot rely on the Cholesky factorization to solve the linear system but instead we have to use some more elaborate techniques such as a QR factorization with pivoting or a singular value decomposition approach which can better handle an almost rank deficient matrix. In our implementation of Algorithm 1, we rely on the routine *dgelsy* from Lapack Anderson et al. (1999), which solves a linear system in the least square sense by using some QR decomposition with pivoting combined with some orthogonalization techniques. Fortunately, the ill-conditioning of the matrix $\boldsymbol{A}$ can be improved by centering and normalizing the polynomials $(B_l)_l$ such that the domain $[0, T] \times \mathcal{D}$ is actually mapped to $[-1, 1]^{d'}$. This reduction improves the numerical behaviour of the chaos decomposition by a great deal.

The construction of the matrix $\boldsymbol{A}$ has a complexity of $O(np^2d')$. The computation of $\boldsymbol{\alpha}$ (Equation 4.3) requires to solve a linear system of size $p \times p$ which requires $O(p^3)$ operations. The overall complexity for computing $\boldsymbol{\alpha}$ is then $O(p^3 + np^2d')$.

Choice of the $(B_l)_l$. The function u^k we want to extrapolate at each step of the algorithm is proved to be quite regular (at least $C^{1,2}$), so using multivariate polynomials for the B_l should provide a satisfactory approximation. Actually, we used polynomials with d' variates, which are built using tensor products of univariate polynomials and if one wants the vector space $\text{Vect}\{B_l,\ l = 1, \dots, p\}$ to be the space of $d'-$variate polynomials with global degree less or equal than η, then p has to be equal to the binomial coefficient $\binom{d'+\eta}{\eta}$. For instance, for $\eta = 3$ and $d' = 6$ we find $p = 84$. This little example shows that p cannot be fixed by specifying the maximum global degree of the polynomials B_l without leading to an explosion of the computational cost, we therefore had to find an other approach. To cope with the curse of dimensionality, we studied different strategies for truncating polynomial chaos expansions. We refer the reader to Chapter 4 of Blatman (2009) for a detailed review on the topic. From a computational point of view, we could not afford the use of adaptive sparse polynomial families because the construction of the family is inevitably sequential and it would have been detrimental for the speed-up of our parallel algorithm. Therefore, we decided to use sparse polynomial chaos approximation based on an hyperbolic set of indices as introduced by Blatman and Sudret (2009).

A canonical polynomial with d' variates can be defined by a multi-index $\nu \in \mathbb{N}^{d'}$ — ν_i being the degree of the polynomial with respect to the variate i. Truncating a polynomial chaos expansion by keeping only the polynomials with total degree not greater than η corresponds to the set of multi-indices: $\{\nu \in \mathbb{N}^{d'} : \sum_{i=1}^{d'} \nu_i \leq \eta\}$. The idea of hyperbolic sets of indices is to consider the pseudo $q-$norm of the multi-index ν with $q \leq 1$

$$\left\{ \nu \in \mathbb{N}^{d'} : \left(\sum_{i=1}^{d'} \nu_i^q \right)^{1/q} \leq \eta \right\}. \tag{3.7}$$

Note that choosing $q = 1$ gives the full family of polynomials with total degree not greater than η. The effect of introducing this pseudo-norm is to favor low-order interactions. Decreasing q leads to a sparser representation, therefore less precise but also faster to compute.

Nb variates	degree	q	size of the basis	reduction factor
3	3	1	20	1
3	3	0.8	13	1.54
3	5	1	56	1
3	5	0.8	35	1.6
3	5	0.6	25	2.24
5	3	1	56	1
5	3	0.8	26	2.15
5	3	0.6	16	3.5
5	5	1	252	1
5	5	0.9	131	1.92
5	5	0.8	96	2.63
5	5	0.6	56	4.5
8	3	1	165	1
8	3	0.8	53	3.11
8	3	0.6	25	6.6
8	5	1	1287	1
8	5	0.9	503	2.55
8	5	0.8	265	4.85

Table 1: Sizes of several hyperbolic bases

4 Parallel approach

In this part, we present a parallel version of the PACVLS algorithm, which is far from being embarrassingly parallel as a crude Monte–Carlo algorithm. We explain the difficulties encountered when parallelizing the algorithm and how we managed to solve them.

4.1 Detailed presentation of the PACVLS algorithm

Here are the notations we use in the algorithm.

- n: number of points of the grid
- k: index of current Picard's iteration
- $\boldsymbol{u}^k = (u^k(t_i^k, x_i^k))_{1\leq i\leq n} \quad \in \mathbb{R}^n$
- $\boldsymbol{c}^k = (c^k(t_i^k, x_i^k))_{1\leq i\leq n} \quad \in \mathbb{R}^n$
- K_{it}: number of iterations of the algorithm
- M: number of Monte–Carlo samples
- N: number of time steps used for the discretization of X
- p: number of functions B_l used in the extrapolating operator. This is not a parameter of the algorithm on its own as it is determined by fixing η and q (the maximum total degree and the parameter of the hyperbolic multi-index set) but the parameter p is of great interest when studying the complexity of the algorithm.

- $(B_l)_{1\leq l\leq p}$ is a family of multivariate polynomials used for extrapolating functions from a finite number of values.
- $\boldsymbol{\alpha}^k \in \mathbb{R}^p$ is the vector of the weights of the chaos decomposition of u^k.
- $d' = d + 1$ is the number of variates of the polynomials B_l.

Algorithm 1 the PACVLS algorithm

$u^0 \equiv 0$, $\boldsymbol{\alpha}^0 \equiv 0$.
for $k = 0 : K_{it} - 1$ **do**
  Pick at random n points $(t_i^k, x_i^k)_{1\leq i\leq n}$.
  for $i = 1 : n$ **do**
    for $m = 1 : M$ **do**
      Let $\boldsymbol{W}$ be a Brownian motion with values in $\mathbb{R}^d$ discretized on a time grid with N time steps.
      Let $U \sim \mathcal{U}_{[0,1]}$.
      Compute

$$a_m^{i,k} = \psi^N\left(t_i^k, x_i^k, f_{u^k} + (\partial_t + \mathcal{L}^N)u^k, \Phi - u^k, \boldsymbol{W}^m, U^m\right).$$

      / *We recall that* $u^k(t,x) = \sum_{l=1}^p \boldsymbol{\alpha}_l^k B_l(t,x)$ */*
    end for

$$\boldsymbol{c}_i^k = \frac{1}{M}\sum_{m=1}^M a_m^{i,k} \tag{4.1}$$

$$\boldsymbol{u}_i^k = \sum_{l=1}^p \boldsymbol{\alpha}_l^k B_l(t_i^k, x_i^k) \tag{4.2}$$

  end for
  Compute

$$\boldsymbol{\alpha}^{k+1} = \arg\min_{\alpha\in\mathbb{R}^p} \sum_{i=1}^n \left|(\boldsymbol{u}_i^k + \boldsymbol{c}_i^k) - \sum_{l=1}^p \alpha_l B_l(t_i^k, x_i^k)\right|^2. \tag{4.3}$$

end for

4.2 Complexity of the algorithm

In this section, we study in details the different parts of Algorithm 1 to determine their complexities. Before diving into the algorithm, we would like to briefly look at the evaluations of the function u^k and its derivatives. We recall that

$$u^k(t,x) = \sum_{l=1}^p \boldsymbol{\alpha}_l^k B_l(t,x)$$

where the $B_l(t,x)$ are of the form $t^{\beta_{l,0}}\prod_{i=1}^d {x_i}^{\beta_{l,i}}$ and the $\beta_{l,i}$ are some integers. Then the computational time for the evaluation of $u^k(t,x)$ is proportional to $p \times d'$. The first and

second derivatives of u^k write

$$\nabla_x u^k(t,x) = \sum_{l=1}^{p} \boldsymbol{\alpha}_l^k \nabla_x B_l(t,x),$$
$$\nabla_x^2 u^k(t,x) = \sum_{l=1}^{p} \boldsymbol{\alpha}_l^k \nabla_x^2 B_l(t,x),$$

and the evaluation of $\nabla_x B_l(t,x)$ (resp. $\nabla_x^2 B_l(t,x)$) has a computational cost proportional to d^2 (resp. d^3).

- The computation (at line 6) of the discretization of the $d-$dimensional Brownian motion with N time steps requires $O(Nd)$ computations.
- The computation of each $a_m^{k,i}$ (line 8) requires the evaluation of the function u^k and its first and second derivatives which has a cost $O(pd^3)$. Then, the computation of $\boldsymbol{c}_i^k$ for given i and k has a complexity of $O(Mpd^3)$.
- The computation of $\boldsymbol{\alpha}$ (Equation 4.3) requires $O(p^3 + np^2d)$ operations as explained in Section 3.2.2.

The overall complexity of Algorithm 1 is $O(K_{it}nM(pd^3 + dN) + K_{it}(p^2nd + p^3))$.

To parallelize an algorithm, the first idea coming to mind is to find loops with independent iterations which could be spread out on different processors with very few communications. The iterations of the outer loop (line 2) are linked from one step to the following, consequently there is no hope parallelizing this loop. On the contrary, the iterations over i (loop line 4) are independent as are the ones over m (loop line 5), so we have at hand two candidates to implement parallelizing. We could even think of a 2 stage parallelism : first parallelizing the loop over i over a small set of processors and inside this first level parallelizing the loop over m. Actually, M is not large enough for the parallelization of the loop over m to be efficient (see Section 3.2). It turns out to be far more efficient to parallelize the loop over i as each iteration of the loop requires a significant amount of work.

4.3 Description of the parallel part

4.3.1 Parallel topology

As we have just explained, we have decided to parallelize the loop over i (line 4 in Algorithm 1). We have used a *Master Slave* approach. In the following, we assume to have $P+1$ processors at hand with $n > P$. We reproduce the following scheme at each step of the algorithm:

1. Send to each of the P slave processors the solution $\boldsymbol{\alpha}^k$ computed at the previous step of the algorithm.
2. Spread the computation of the vector $\boldsymbol{c}^k$ among the P slave processors.
3. Get from all the slave processors their contributions to the computation of $\boldsymbol{c}^k$ and compute $\boldsymbol{\alpha}^{k+1}$. This part is done by the master process.

Note that at the end of every iteration, ie. after the computation the vector $\boldsymbol{\alpha}$, all the processors have to be synchronised, which is obviously a bottle neck of our approach but also an intrinsic characteristic of the mathematical problem. Now, we make precise how we have implemented step 2 of the above scheme

Load–balancing. At step k, the computation of the vector $\boldsymbol{c}^k$ requires to run a Monte–Carlo computation at each point of the grid $(t_i^k, x_i^k)_{1\leq i\leq n}$. The burning issue is to know how to distribute these computations among the P slave processes. Several strategies can be used to do so, we have decided to concentrate on two approaches: the first one is fully dynamic while the second is purely static.

- **Dynamic approach.** At iteration k, the master assigns slave i the computation of the correction term $(\boldsymbol{c}_i^k)$ for all $1 \leq i \leq P$. As soon as a slave process has finished its computation, it sends the result back and the master process assigns it the computation of the correction $\boldsymbol{c}^k$ at an other point of the grid. The process goes on until the correction has been computed at all points. In this approach, the approximate number of communications between the master and a slave process is $\lfloor n/P \rfloor + 1$.

- **Static approach.** Before starting any computations, assign to each process a block of points at which the corrections $\boldsymbol{c}^k$ should be computed and send each slave process the corresponding data all at once. This way, at each iteration k, only two communications between the master and a slave process have to be initialised : one at the beginning to send the data and one at the end to get the result back.

The performances of these two strategies are compared in details in Section 6.3.

Passing Messages. Considering the wide range of data to be sent and the intensive use of complex structures, the most natural way to pass these objects was to rely on the packing mechanism of MPI. Moreover, packing enables to build a message composed of objects of different types, which can therefore be passed in a single message. The numerical library we are using in the code (see Section 4.3.3) already has a MPI binding which makes the packing mechanism and message passing almost transparent.

4.3.2 Random numbers in a parallel environment

One of the basic problem when solving a probabilistic problem in parallel computing is the generation of random numbers. Random number generators are usually devised for sequential use only and special care should be taken in parallel environments to ensure that the sequences of random numbers generated on each processor are independent. We would like to have minimal communications between the different random number generators, ideally after the initialisation process, each generator should live independently of the others.

There are basically two strategies for that : either to split a unique stream in substream or to create independent streams.

1. Splitting a sequence of random numbers across several processors can only be efficiently implemented if the generator has some splitting facilities such that there is no need to draw all the samples prior to any computations. We refer to L'Ecuyer and Côté (1991); L'Ecuyer et al. (2002) for a presentation of a generator with splitting facilities.

To efficiently split the sequence, one should know in advance the number of samples needed by each processor or at least an upper bound of it. To encounter this problem, the splitting could be made in substreams by jumping ahead of P steps at each call to the random procedure if P is the number of processors involved. This way, each processor uses a sub-sequence of the initial random number sequence rather than a contiguous part of it. However, as noted by Entacher et al. (1999), long range correlations in the original sequence can become short range correlations between different processors when using substreams.

Actually, the best way to implement splitting is to use a generator with a huge period such as the Mersenne Twister (its period is $2^{19937}-1$) and divide the period by a million or so if we think we will not need more than a million independent substreams. Doing so, we come up with substreams which still have an impressive length, in the case of the Mersenne Twister each substream is still about 2^{19917} long.

2. A totally different approach is to find generators which can be easily parametrised and to compute sets of parameters ensuring the statistical independence of the related generators. Several generators offer such a facility such as the ones included in the SPRNG package (see Mascagni (1997) for a detailed presentation of the generators implemented in this package) or the dynamically created Mersenne Twister (DCMT in short), see Matsumoto and Nishimura (2000).

For our experiments, we have decided to use the DCMT. This generator has a sufficiently long period (2^{521} for the version we used) and we can create at most $2^{16} = 65536$ independent generators with this period which is definitely enough for our needs. Moreover, the dynamic creation of the generators follows a deterministic process (if we use the same seeds) which makes it reproducible.

4.3.3 The library used for the implementation

Our code has been implemented in C using the PNL library (see Lelong (2007-2011)). This is a scientific library available under the Lesser General Public Licence and it offers various facilities for solving mathematical problems and more recently some MPI bindings have been added to easily manipulate the different objects available in PNL. In our problem, we needed to manipulate matrices and vectors and pass them from the master process to the slave processes and decided to use the packing facilities offered by PNL through its MPI binding. The technical part was not only message passing but also random number generation as we already mentioned above and PNL offers many functions to generate random vectors or matrices using several random number generators among which the DCMT.

Besides message passing, the algorithm also requires many other facilities such as multivariate polynomial chaos decomposition which is part of the library. For the moment, three families of polynomials (Canonical, Hermite and Tchebichev polynomials) are implemented along with very efficient mechanism to compute their first and second derivatives. The implementation tries to make the most of code factorization to avoid recomputing common quantities several times. The polynomial chaos decomposition toolbox is quite flexible and offers a reduction facility such as described in Section 3.2.2 which is completely transparent from the user's side. To face the curse of dimensionality, we used sparse polynomial families based on an hyperbolic set of indices.

5 Numerical experiments

In this section, we study the convergence of the PACVLS Algorithm and test its parallel version in high dimension. To do so, we compare our results with benchmarks ensuing from financial problems. Then, we apply the parallel version of the PACVLS Algorithm to price and hedge European options. We consider the case of the Black-Scholes model with a linear driver (through the pricing of European options in a standard case) and with a non linear driver (through the pricing of European options with a borrowing rate higher than the bond rate).

5.1 Framework

Let $(W_t)_{t\geq 0}$ be a standard Brownian motion with values in $\mathbb{R}^d$. We denote by $(\mathcal{F}_t)_{t\geq 0}$ the $\mathbb{P}$-completion of the natural filtration of $(W_t)_{t\geq 0}$.
Consider a financial market with a risk-free asset satisfying $dS_t^0 = rS_t^0 dt$ and d risky assets, with prices $S_t^1, \cdots, S_t^d$ at time t. We assume that (S_t) satisfies the following stochastic differential equation:

$$dS_t^i = S_t^i \left(\mu^i dt + \sum_{j=1}^{d} \Sigma^{ij} dW_t^j \right), \ i = 1, \cdots, d \tag{5.1}$$

on a finite interval $[0, T]$, where T is the maturity of the option, μ^i represents the trend of S^i and $(\Sigma^{ij})_{uj}$ is the matrix of volatility which embeds both the correlation between the assets and the volatilities of each of them. We denote by $S_s^{t,x}$ a continuous version of the flow of the stochastic differential Equation (5.1). $S_t^{t,x} = x$ almost surely.

We are interested in computing the price of a European option with payoff $\Phi(S_T)$, where $\Phi : \mathbb{R}^d \longmapsto \mathbb{R}_+$ is a continuous function and S follows (5.1).

Linear driver We denote by V_t the option price and by π_t the amount of the wealth V_t invested in the ith stock at time t. From El Karoui et al. (1997), we know that the couple (V, π) satisfies

$$-dV_t = rV_t dt + \pi_t^* \Sigma \theta dt + \pi_t^* \Sigma dW_t, V_T = \Phi(S_T),$$

where θ is the solution (supposed to be unique) of the linear system $\mu - r\mathbf{1} = \Sigma\theta$ where $\mathbf{1}$ is a vector whose elements are all 1. Then, (V, π) is solution of a standard BSDE (1.1). Y corresponds to V, Z corresponds to $\pi_t^*\Sigma$ and the driver $f(t, x, y, z) := -ry - z\theta$.

Non Linear driver Let us now consider the hedging of claims with a borrowing rate higher than the bond rate. We refer to El Karoui et al. (1997) for this example of constrained portfolio. We consider the case where the investor is allowed to borrow money at time t at an interest rate $R > r$, where r is the constant bond rate. We borrow and invest money in the bond at the same time, but we restrict ourselves to policies in which the amount borrowed at time t is equal to $(V_t - \sum_{i=1}^d \pi_t^i)^-$. The strategy (wealth, portfolio) (V, π) satisfies

$$dV_t = rV_t dt + \pi_t^* \Sigma \theta dt + \pi_t^* \Sigma dW_t - (R - r) \left(V_t - \sum_{i=1}^{d} \pi_t^i \right)^- dt.$$

Finding the strategy (V, π) consists in solving BSDE (1.1) with the nonlinear driver $f(t, x, y, z) := -ry - z\theta + (R - r)(y - \sum_{i=1}^d (z\Sigma^{-1})_i)^-$.

5.2 Numerical results

The accuracy tests have been achieved using the facilities offered by the University of Savoie computing center MUST.

We consider a $d-$dimensional Black-Scholes model in which the dynamics under the risk neutral measure of each asset S^i is supposed to be given by

$$dS_t^i = S_t^i(\mu^i dt + \sigma^i dB_t^i) \qquad S_0 = (S_0^1, \dots, S_0^d) \tag{5.2}$$

where $B = (B^1, \dots, B^d)$. We deliberately keep μ^i in the definition of S^i (instead of replacing it by r as we usually do) to keep the dependency on z in f. The vector $\sigma = (\sigma^1, \dots, \sigma^d)$ is the vector of volatilities.

Each component B^i is a standard Brownian motion. For the numerical experiments, the covariance structure of B will be assumed to be given by $\langle B^i, B^j \rangle_t = \rho t \mathbf{1}_{\{i \neq j\}} + t\mathbf{1}_{\{i=j\}}$. We suppose that $\rho \in (-\frac{1}{d-1}, 1)$, which ensures that the matrix $C = (\rho \mathbf{1}_{\{i\neq j\}} + \mathbf{1}_{\{i=j\}})_{1\leq i,j\leq d}$ is positive definite. Let L denote the lower triangular matrix involved in the Cholesky decomposition $C = LL^*$. We have $(B_t)_t \overset{\text{law}}{=} (LW_t)_t$, where is the Brownian motion introduced in (5.1).

Since we know how to simulate the law of (S_t, S_T) exactly for $t < T$, there is no use to discretize equation (5.2) using the Euler scheme. In this Section $N = 2$.

We want to study the numerical accuracy of our algorithm and to do that we consider the case of European basket options for which we can compute benchmark price by using very efficient Monte-Carlo methods, see Jourdain and Lelong (2009) for instance.

Standard European put basket option (linear driver). Consider the following put basket option with maturity T

$$\left(K - \frac{1}{d}\sum_{i=1}^{d} S_T^i\right)_+ \tag{5.3}$$

in the standard case, i.e. when the borrowing rate equals the bond one.

Figure 1 presents the influence of the parameters M and n. The results obtained for $M = 50,000$ (curves $(\times)$ and $(*)$) are very close to the true price, for both values of n. Moreover we can see that the algorithm stabilizes after very few iterations (less than 5). The result obtained for $M = 5,000$ (curve $(+)$) does not converge.

To conclude, we notice that the larger the number of Monte–Carlo simulations is, the smoother the convergence is. The influence of n does not seem crucial.

Standard European call basket option (linear driver). Consider the following call basket option with maturity T

$$\left(\frac{1}{d}\sum_{i=1}^{d} S_T^i - K\right)_+ \tag{5.4}$$

in the standard case, i.e. when the borrowing rate equals the bond one. Figure 2 represents the evolution of the approximated price of a European call option in dimension 8 for $M = 5000$ (curve $(+)$) and for $M = 50,000$ (curve $(\times)$). As for the pricing of a put basket option, one notices that the curve $(\times)$ converges very fast to the reference price. This curve corresponds

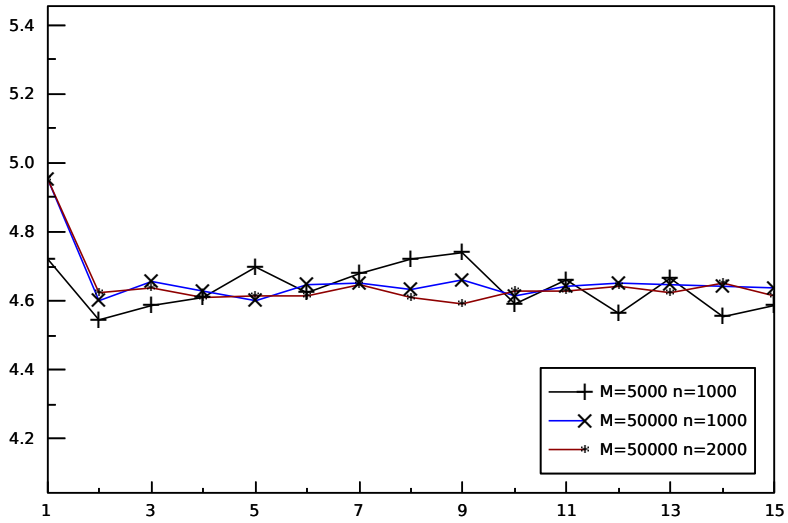


Figure 1: Convergence of the algorithm for a European put basket option with $d = 5$, $\rho = 0.1$, $T = 3$, $S_0 = 100$, $\sigma^i = 0.2$, $\mu^i = 0.02$, $r = 0.02$ $K = 100$, $\eta = 3$, $q = 1$. The benchmark price computed with a high precision Monte–Carlo method yields 4.6440 with a confidence interval of $(4.6415, 4.6466)$.

to $M = 50,000$ and $n = 1,000$. Curve $(+)$ (M=5,000) oscillates around the reference price. As before, one notices that the larger M is, the faster the convergence is, even for small values of n.

European put basket option with higher borrowing rate (non linear driver). Consider the put basket option with payoff given by (5.3) with maturity T when the borrowing rate differs from the bond one. In such a case, the driver is non linear and satisfies $f(t, x, y, z) := -ry - z\theta + (R - r)(y - \sum_{i=1}^{d}(z\Sigma^{-1})_i)^-$. In this case we do not have any reference price, we only study the convergence of the algorithm.

Figure 3 illustrates the impact of the sparsity of the polynomial basis on the convergence of the algorithm. The fastest convergence is achieved for the curve $(\times)$, i.e. when $\eta = 3$ and $q = 1$. The algorithm stabilizes near 1.61 after very few iterations.

- Influence of η : for a fixed value of q, the sparsity increases when η decreases, so the basis with $\eta = 3, q = 0.8$ is more sparse than the one with $\eta = 5, q = 0.8$. When comparing curves $(*)$ $(\eta = 5)$ and $(+)$ $(\eta = 3)$ for fixed values of q $(= 0.8)$, we can see that for $\eta = 3$ (curve $(+)$) the algorithm converges to 1.4, whereas for $\eta = 5$ (curve $(*)$) the algorithm converges to 1.57. The higher is η, the better is the convergence.

- Influence of q : for fixed values of η $(= 5)$, we compare curves $(*)$ $(q = 0.8)$ and $(\blacklozenge)$ $(q = 0.9)$. We can see that for $q = 0.9$ (curve $(\blacklozenge)$) the algorithm converges nearer to 1.61 than for $q = 0.8$ (curve $(*)$).

Actually, when the polynomial basis becomes too sparse, the approximation of the solution computed at each step of the algorithm incorporates a significant amount of noise which has

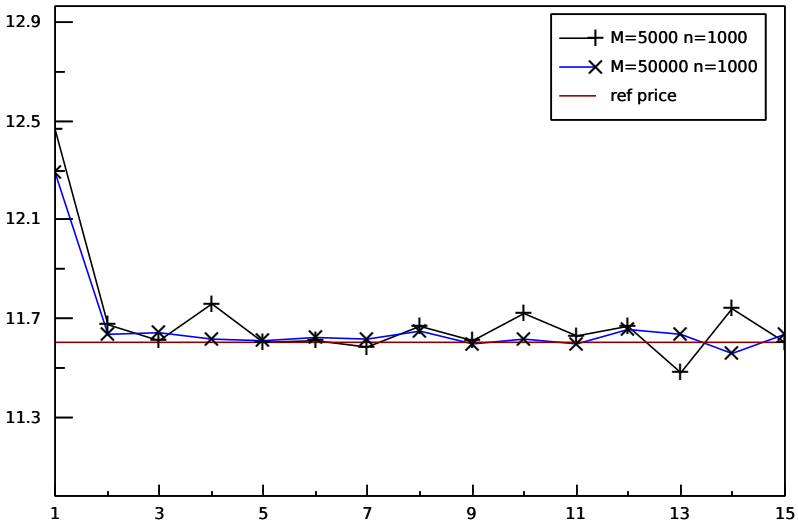


Figure 2: Convergence of the price of a European call basket option with $d = 8$, $\rho = 0.3$, $T = 3$, $S_0 = 100$, $\sigma^i = 0.2$, $r = 0.02$, $K = 100$, $\eta = 3$, $q = 1$. The benchmark price computed with a high precision Monte–Carlo method yields 11.6041 with a confidence interval of $(11.5994, 11.6088)$.

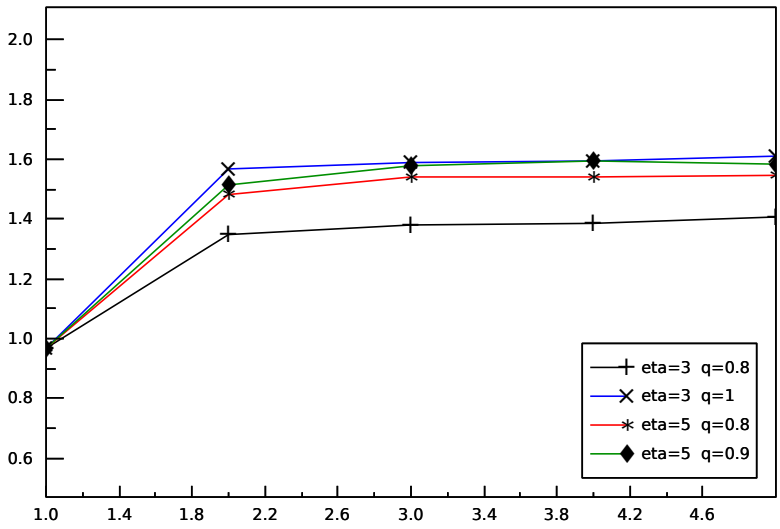


Figure 3: Convergence of the price of a European put basket option with $d = 5$, $\rho = 0.1$, $T = 1$, $S_0 = 100$, $\sigma^i = 0.2$, $\mu^i = 0.05$, $R = 0.1$, $r = 0.02$ $K = 95$, $M = 30,000$ and $n = 2000$.

a similar effect to reducing the number of Monte–Carlo simulations. This is precisely what we observe on Figure 4: the curves $(\times)$ $(M = 30,000,\ n = 2,000,\ \eta = 5,\ q = 0.8)$ and $(*)$

($M = 5,000$, $n = 1,000$, $\eta = 5$, $q = 0.9$) have very similar behaviours although curve ($\times$) has a much larger number of simulations.

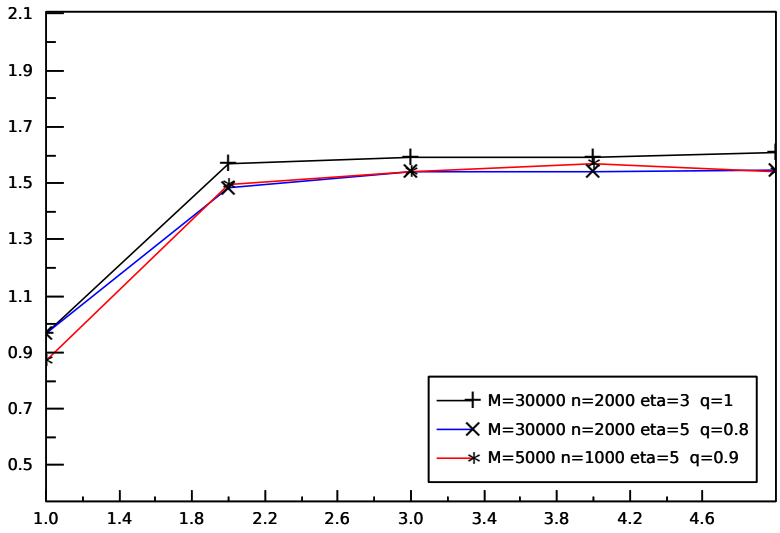


Figure 4: Convergence of the price of a European put basket option with $d = 5$, $\rho = 0.1$, $T = 1$, $S_0 = 100$, $\sigma^i = 0.2$, $\mu^i = 0.05$, $R = 0.1$, $r = 0.02$, $K = 95$.

6 Performance analysis

6.1 The cluster

All our performance tests have been carried out on a PC cluster from INRIA Paris–Rocquencourt with 316 cores. Each node has two processors with six cores per processor: INTEL Xeon X5650 2.67 GHz. Inside one node, all the cores share 48Gb of RAM. All the nodes are interconnected using a Gigabit Ethernet network. In none of the experiments, did we make the most of the multi core architecture since our code is single threaded. Hence, in our implementation a multi core processor is actually seen as many single core processors.

6.2 The performance measurements

The performance analysis of the algorithm has been carried out in the Black–Scholes model in dimensions 5 and 8.

We summarize in Tables 2 and 3 all the computational time measures we have performed on our algorithm. The computational times are expressed in seconds. We show on Figures 5 and 6 our speedup measures as cross marks "+". One may wonder why the speedup graphs are so irregular; there are two reasons for that. First, since the algorithm is random, the computational time may vary slightly between two runs. Second, adding one more processor only decreases the computation time if it enables to reduce the amount of computations of the most loaded processors and this is not so frequent. Let us take an example : assume we

have $2,000$ correction terms to compute and 100 slave processors, each of them will compute 20 correction terms; but if instead we have 101 slave processors, 20 slave processors will compute 19 correction terms whereas the 81 other slave processors will still have to compute 20 correction terms, hence the time to wait for the result will be the same for 100 and 101 processors. We need at least 106 processors to eventually reduce the computational time; in this case, each slave processor will compute at most 19 correction terms.

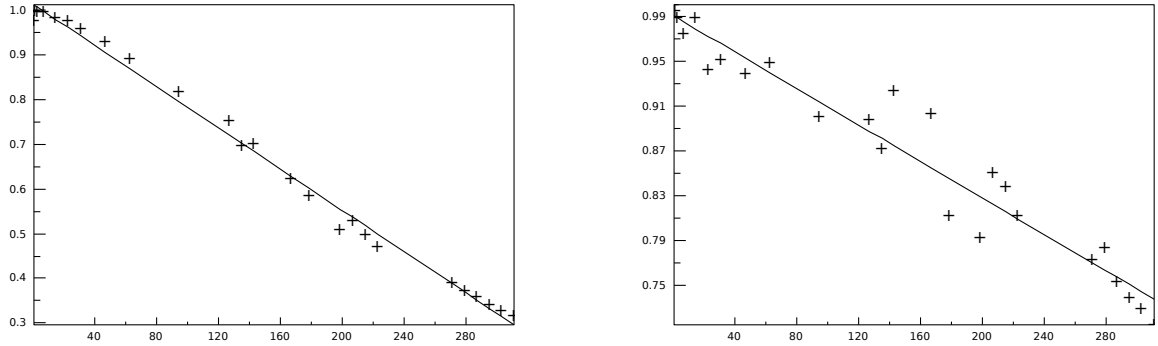
Therefore, we thought that an average speedup function could be more meaningful and we decided to plot the linear regression of the speedup measures as plain lines on our graphs (see Figures 5 and 6).

Nb proc.	Time	Speed up
1	12223	0.977647
2	5988	0.997727
3	3999	0.995902
7	1711	0.997226
15	810	0.983462
23	531	0.977149
31	402	0.958182
47	273	0.929001
63	212	0.891802
95	153	0.817773
127	125	0.751972
135	126	0.697084
143	119	0.701623
167	114	0.624322
179	114	0.584023
199	118	0.508844
207	109	0.527103
215	111	0.497231
223	113	0.470531
271	113	0.389894
279	115	0.371787
287	116	0.357697
295	118	0.341882
303	120	0.327936
311	122	0.314424

Nb proc.	Time	Speed up
1	11954	0.99961
2	6010	0.994146
3	4030	0.988221
7	1752	0.974141
15	806	0.988106
23	551	0.941649
31	405	0.950102
47	271	0.937632
63	199	0.94874
95	139	0.89962
127	104	0.896829
135	101	0.871769
143	90	0.92315
167	79	0.902976
179	82	0.811206
199	75	0.792396
207	67	0.849704
215	66	0.83697
223	66	0.811259
271	57	0.772816
279	54	0.783046
287	55	0.753074
295	54	0.738246
303	54	0.72895
311	53	0.714815

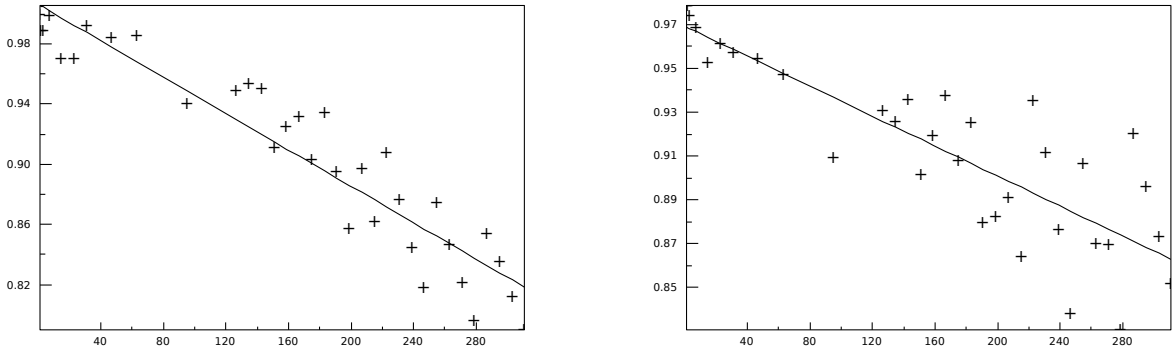
Table 2: Speed up in dimension 5 using the dynamic (resp. static) load balancing strategy in the left (resp. right) table

A quick comparison of Figures 5 and 6 shows that the scalability of our approach becomes better when the dimension of the problem increases. We expected such a behaviour as the computational cost increases much faster that the communication cost. Actually, the dynamic load–balancing approach is far more sensitive to the dimension of the problem than the static one: from our experiments, the static approach always outperforms the dynamic one. We refer to Section 6.3 for a detailed analysis of the two strategies.



(a) dynamic load–balancing (b) static load–balancing

Figure 5: Speedups in dimension 5



(a) dynamic load–balancing (b) static load–balancing

Figure 6: Speedups in dimension 8

6.3 Load–balancing

From a theoretical point of view, the dynamic load–balancing ensures that the computations are better shared among the nodes such that all the nodes ideally stop working at the same time. This dynamic approach should show better results than the static one, but one has to take into account the extra communications induced by the dynamism. Actually, we can see from Figures 5 and 6 that our experiments contradict this theoretical intuition and we will try to understand why. To do so, we first analyze the static approach and study how we could improve it.

Static approach. The idea of the static approach is to equally spread the computations of the n correction terms $(\boldsymbol{c}_i^k)_{1\leq i\leq n}$. Practically, there is hardly no chance that the number of processors P divides n; hence, one has to be careful of how to treat the remainder since the computation cost of one correction term is far from negligible. Therefore, the remaining

Nb proc.	Time	Speed up
1	145566	0.999546
2	73590	0.988585
3	49060	0.98857
7	20810	0.998816
15	9998	0.970118
23	6518	0.970422
31	4730	0.992248
47	3145	0.984333
63	2342	0.985792
95	1628	0.940501
127	1207	0.948543
135	1130	0.953726
143	1070	0.950526
151	1057	0.911502
159	989	0.924526
167	934	0.932029
175	920	0.902949
183	851	0.93369
191	851	0.894721
199	852	0.857354
207	783	0.896722
215	785	0.861675
223	718	0.907823
231	718	0.876125
239	720	0.844622
247	720	0.817644
255	652	0.874123
263	653	0.846656
271	654	0.820905
279	655	0.795728
287	593	0.853546
295	590	0.835287
303	591	0.811958
311	592	0.789684

Nb proc.	Time	Speed up
1	148655	0.978776
2	74355	0.978414
3	49817	0.973548
7	21465	0.968314
15	10183	0.952539
23	6580	0.961336
31	4903	0.95723
47	3243	0.954376
63	2438	0.946965
95	1684	0.909315
127	1231	0.93038
135	1164	0.925512
143	1087	0.935746
151	1068	0.901408
159	995	0.919115
167	929	0.937534
175	915	0.907674
183	859	0.92505
191	866	0.879222
199	828	0.882184
207	788	0.890912
215	783	0.863829
223	697	0.935231
231	691	0.911311
239	694	0.876406
247	702	0.838182
255	629	0.906539
263	635	0.870393
271	617	0.869531
279	627	0.830648
287	550	0.920515
295	550	0.896195
303	549	0.873166
311	549	0.851524

Table 3: Speed up in dimension 8 using the dynamic (resp. static) load balancing strategy in the left (resp. right) table

computations ($n - \lfloor n/P \rfloor \times P$ correction terms to be computed) are uniformly shared among all the processors, which means that some processors compute $\lfloor n/P \rfloor + 1$ correction terms whereas the others only compute $\lfloor n/P \rfloor$ terms. Obviously, in this approach we assume that all the correction terms require the same computational effort, which is a realistic assumption at least at large scale. This computation effort may vary depending on the $(t_i)_{i \leq i \leq n}$ when using the Euler scheme; however if we assume that the number of t_i is large enough, then the

strong law of large numbers guaranties that the computational effort should be roughly the same for all the processors.

Dynamic approach. In the dynamic load–balancing, at the beginning of every iteration, the master process entrusts every slave process with the computation of one correction term; this means that the master has to communicate with each slave. Then, as soon as a slave finishes its computation, it sends the result back to the master process which in turn sends it back a new correction term to be computed and the process goes on until all the correction terms are computed. Obviously, this way of balancing computations looks smart but it creates far more communications than in the static approach. This is actually confirmed by our experiments in which the static approach is faster and shows a better scalability. The dynamic approach requires $2n$ communications between the master and a slave process whereas the static one requires only $2P$ communications. To improve the scalability of the dynamic approach, we could try to use a divide to conquer approach: we could group corrections and consider several master processes, each of them being in charge of balancing a bunch of computations. This would significantly reduce the overhead between the computations of two correction terms.

7 Conclusion

In this work, we have presented a parallel algorithm for solving BSDE in high dimensions and applied it to the pricing and hedging of European options with a bond rate different from the borrow rate. Solving a BSDE at large scale remains a computationally demanding problem for which very few scalable implementations have been studied. Our parallel algorithm shows an encouraging scalability in high dimensions. To improve the efficiency of the algorithm, we could try to refactor the extrapolation step to make it more accurate and less sensitive to the curse of dimensionality. The extrapolation step is solved sequentially for the moment and a first improvement could be to use a multi–thread solver for this part.

References

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. ISBN 0-89871-447-8 (paperback).

V. Bally and G. Pagès. Error analysis of the optimal quantization algorithm for obstacle problems. *Stochastic Processes and their Applications*, 106(1):1–40, 2003.

C. Bender and R. Denk. A forward scheme for backward SDEs. *Stochastic Processes and their Applications*, 117(12):1793–1812, 2007.

G. Blatman. *Adaptive sparse polynomial chaos expansions for uncertainty propagation and sensitivity analysis*. PhD thesis, Université Blaise Pascal - Clermont II, 2009.

G. Blatman and B. Sudret. Anisotropic parcimonious polynomial chaos expansions based on the sparsity-f-effects principle. In *Proc ICOSSAR'09, International Conference in Structural Safety and Relability*, 2009.

B. Bouchard and N. Touzi. Discrete time approximation and Monte Carlo simulation of backward stochastic differential equations. *Stochastic Processes and their Applications*, 111:175–206, 2004.

F. Delarue and S. Menozzi. A forward-backward stochastic algorithm for quasi-linear PDEs. *Annals of Applied Probability*, 16(1):140–184, 2006.

N. El Karoui, S. Peng, and M. Quenez. Backward Stochastic Differential Equations in Finance. *Mathematical Finance*, 7(1):1–71, 1997.

K. Entacher, A. Uhl, and S. Wegenkittl. Parallel random number generation: Long-range correlations among multiple processors, 1999.

E. Gobet and C. Labart. Solving BSDE with adaptive control variate. *SIAM Journal of Num. Anal.*, 48(1), 2010.

E. Gobet, J. Lemor, and X. Warin. A regression-based Monte Carlo method to solve backward stochastic differential equations. *Annals of Applied Probability*, 15(3):2172–2202, 2005.

B. Jourdain and J. Lelong. Robust adaptive importance sampling for normal random vectors. *Annals of Applied Probability*, 19(5):1687–1718, 2009.

P. L'Ecuyer and S. Côté. Implementing a random number package with splitting facilities. *ACM Trans. Math. Softw.*, 17(1):98–111, 1991. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/103147.103158.

P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Oper. Res.*, 50(6):1073–1075, 2002. ISSN 0030-364X. doi: http://dx.doi.org/10.1287/opre.50.6.1073.358.

J. Lelong. Pnl. `http://www-ljk.imag.fr/membres/Jerome.Lelong/soft/pnl/index.html`, 2007-2011.

J. Ma, P. Protter, and J. Yong. Solving forward backward stochastic differential equations explicitly-a four step scheme. *Probability Theory Related Fields*, 98(1):339–359, 1994.

M. Mascagni. Some methods of parallel pseudorandom number generation. In *in Proceedings of the IMA Workshop on Algorithms for Parallel Processing*, pages 277–288. Springer Verlag, 1997. available at `http://www.cs.fsu.edu/~mascagni/papers/RCEV1997.pdf`.

M. Matsumoto and T. Nishimura. *Monte Carlo and Quasi-Monte Carlo Methods 1998*, chapter Dynamic Creation of Pseudorandom Number Generator. Springer, 2000. available at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dgene.pdf`.

E. Pardoux and S. Peng. Backward Stochastic Differential Equations and Quasilinear Parabolic Partial Differential Equations. *Lecture Notes in CIS*, 176(1):200–217, 1992.

Y. Wang and R. Caflish. Pricing and hedging american-style options: a simple simulation-based approach. *The Journal of Computational Finance*, 13(3), 2010.