A Parallel Algorithm for solving BSDEs - Application to the pricing and hedging of American options

Céline Labart [*]
Laboratoire de Mathématiques, CNRS UMR 5127
Université de Savoie, Campus Scientifique
73376 Le Bourget du Lac, France

Jérôme Lelong [†]
Laboratoire Jean Kuntzmann,
Université de Grenoble and CNRS,
BP 53, 38041 Grenoble, Cedex 09, France

February 23, 2011

Abstract

We present a parallel algorithm for solving backward stochastic differential equations (BSDEs in short) which are very useful theoretic tools to deal with many financial problems ranging from option pricing option to risk management. Our algorithm based on Gobet and Labart (2010) exploits the link between BSDEs and non linear partial differential equations (PDEs in short) and hence enables to solve high dimensional non linear PDEs. In this work, we apply it to the pricing and hedging of American options in high dimensional local volatility models, which remains very computationally demanding. We have tested our algorithm up to dimension 10 on a cluster of 512 CPUs and we obtained linear speedups which proves the scalability of our implementation.

Keywords : backward stochastic differential equations, parallel computing, Monte-Carlo methods, non linear PDE, American options, local volatility model.

1 Introduction

Pricing and hedging American options with a large number of underlying assets is a challenging financial issue. On a single processor system, it can require several hours of computation in high dimensions. Recent advances in parallel computing hardware such as multi–core processors, clusters and GPUs are then of high interest for the finance community. For a couple of years, some research teams have been tackling the parallelization of numerical algorithms for option pricing. Thulasiram and Bondarenko (2002) developed a parallel algorithm using MPI for pricing a class of multidimensional

[*]Email:celine.labart@univ-savoie.fr
[†]Email:jerome.lelong@imag.fr

financial derivatives using a binomial lattice approach. Huang and Thularisam (2005) presented algorithms for pricing American style Asian options using a binomial tree method. Concerning the parallelization of Monte–Carlo methods for pricing multi–dimensional Bermudan/American options, the literature is quite rare. We refer to Toke and Girard (2006) and to Dung Doan et al. (2010). Both papers propose a parallelization through grid computing of the Ibáñez and Zapatero (2004) algorithm, which computes the optimal exercise boundary. A GPGPU approach based on quantization techniques has recently been developed by Pagès and Wilbertz (2011).

Our approach is based on solving Backward Stochastic Differential Equations (BSDEs in short). As explained in the seminal paper by El Karoui et al. (1997b), pricing and hedging European options in local volatility models boil down to solving standard BSDEs. From El Karoui et al. (1997a), we know that the price of an American option is also linked to a particular class of BSDEs called reflected BSDEs. Several sequential algorithms to solve BSDEs can be found in the literature. Ma et al. (1994) adopted a PDE approach, whereas Bouchard and Touzi (2004) and Gobet et al. (2005) used a Monte–Carlo approach based on the dynamic programming equation. The Monte–Carlo approach was also investigated by Bally and Pagès (2003) and Delarue and Menozzi (2006) who applied quantization techniques to solve the dynamic programming equation. Our approach is based on the algorithm developed by Gobet and Labart (2010) which combines Picard's iterations and an adaptive control variate. It enables to solve standard BSDEs, ie, to get the price and delta of European options in a local volatility model. Compared to the algorithms based on the dynamic programming equation, ours provides regular solutions in time and space (which is coherent with the regularity of the option price and delta). To apply it to the pricing and hedging of American options, we use a technique introduced by El Karoui et al. (1997a), which consists in approximating a reflected BSDE by a sequence of standard BSDEs with penalisation.

The paper is organized as follows. In section 2, we briefly recall the link between BSDEs and PDEs which is the heart of our algorithm. In section 3, we describe the algorithm and in Section 4 we explain how the parallelization has been carried out. Section 5 describes how American options can be priced using BSDEs. Finally, in Section 6, we conclude the paper by some numerical tests of our parallel algorithm for pricing and hedging European and American basket options in dimension up to 10.

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 BSDEs

2.1 General results on standard BSDEs

Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a given probability space on which is defined a q-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 (Y, Z) the solution of the following backward stochastic differential equation (BSDE) with fixed terminal time T

$$-dY_t = f(t, X_t, Y_t, Z_t)dt - Z_t dW_t, \quad Y_T = \Phi(X_T), \tag{2.1}$$

where $f : [0, T] \times \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^q \to \mathbb{R}$, $\Phi : \mathbb{R}^d \to \mathbb{R}$ and X is the $\mathbb{R}^d$-valued process solution of

$$X_t = x + \int_0^t b(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s, \tag{2.2}$$

with $b : [0, T] \times \mathbb{R}^d \to \mathbb{R}^d$ and $\sigma : [0, T] \times \mathbb{R}^d \to \mathbb{R}^{d\times q}$.

From now on, we assume the following Hypothesis, which ensures the existence and uniqueness of the solution to Equations (2.1)-(2.2).

Hypothesis 1

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

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

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

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

- *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 $C_b^{0,1}$.*

2.2 Link with semilinear PDEs

Let us also recall the link between BSDEs and semilinear PDEs. Although the relation is the keystone of our algorithm, as explained in Section 3, we do not develop it and refer to Pardoux and Peng (1992) or El Karoui et al. (1997b) for more details.

According to Pardoux and Peng (1992, Theorem 3.1), we can link the solution (Y, Z) of the BSDE (2.1) to the solution u of the following PDE:

$$\begin{cases} \partial_t u(t, x) + \mathcal{L}u(t, x) + f(t, x, u(t, x), (\partial_x u\sigma)(t, x)) = 0, \\ u(T, x) = \Phi(x), \end{cases} \tag{2.3}$$

where $\mathcal{L}$ is defined by

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

Theorem 1 (Delarue and Menozzi (2006), Theorem 2.1)**.** *Under Hypothesis 1, the solution u of PDE (2.3) belongs to $C_b^{1,2}$. Moreover, the solution $(Y_t, Z_t)_{0\leq t\leq T}$ of (2.1) satisfies*

$$\forall t \in [0,T], \ \ (Y_t, Z_t) = (u(t, X_t), \partial_x u(t, X_t)\sigma(t, X_t)). \tag{2.4}$$

3 Presentation of the Algorithm

3.1 Description

We present the algorithm introduced by Gobet and Labart (2010) to solve standard BSDEs. It is based on Picard's iterations combined with an adaptive Monte–Carlo method. We recall that we aim at numerically solving BSDE (2.1), which is equivalent to solving the semilinear PDE (2.3). The current algorithm provides an approximation of the solution of this PDE. Let u^k denote the approximation of the solution u of (2.3) at step k. If we are able to compute an explicit solution of (2.2), the approximation of (Y, Z) at step k follows from (2.4): $(Y^k_t, Z^k_t) = (u^k(t, X_t), \partial_x u^k(t, X_t)\sigma(t, X_t))$, for all $t \in [0, T]$. Otherwise, we introduce X^N the approximation of X obtained with a N–time step Euler scheme:

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

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

$$(Y^k_t, Z^k_t) = (u^k(t, X^N_t), \partial_x u^k(t, X^N_t)\sigma(t, X^N_t)), \text{ for all } t \in [0, T].$$

It remains to explain how to build the approximation $(u^k)_k$ of u. The basic idea is the following:

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

Combining Itô's formula applied to $u(s, X_s)$ and $u^k(s, X^N_s)$ between t and T and the semilinear PDE (2.3) satisfied by u, we get that the correction term c^k is given by

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

where

- $\mathcal{L}^N u(s, X^N_s) = \frac{1}{2}\sum_{i,j}[\sigma\sigma^*]_{ij}(\varphi(s), X^N_{\varphi(s)})\partial^2_{x_i x_j}u(s, X^N_s) + \sum_i b_i(\varphi(s), X^N_{\varphi(s)})\partial_{x_i}u(s, X^N_s)$.
- $f_v : [0, T] \times \mathbb{R}^d \to \mathbb{R}$ denotes the following function

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

  where f is the driver of BSDE (2.1), 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

$$\Psi(s,y,g_1,g_2,W) = \int_s^T g_1(r, X^{s,y}_r(W))dr + g_2(X^{s,y}_T(W)),$$

$$\Psi^N(s,y,g_1,g_2,W) = \int_s^T g_1(r, X^{N,s,y}_r(W))dr + g_2(X^{N,s,y}_T(W)),$$

where $X^{s,y}$ (resp. $X^{N,s,y}$) denotes the diffusion process solving (2.2) and starting from y at time s (resp. its approximation using an Euler scheme with N time steps), and W denotes the standard Brownian motion appearing in (2.2) and used to simulate X^N, as given in (3.1).

- $\mathcal{G}^k$ is the σ-algebra generated by the set of all random variables used to build u^k. In the above formula of c^k, we compute the expectation w.r.t. the law of X and X^N and not w.r.t. the law of u^k, which is $\mathcal{G}^k$ measurable. (See Definition 2 for a rigorous definition of $\mathcal{G}^k$).

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

$$\Psi(s,y,g_1,g_2,W) = \mathbb{E}_U\left[(T-s)g_1(s+(T-s)U, X^{s,y}_{s+(T-s)U}(W)) + g_2(X^{s,y}_T(W))\right],$$
$$\Psi^N(s,y,g_1,g_2,W) = \mathbb{E}_U\left[(T-s)g_1(s+(T-s)U, X^{N,s,y}_{s+(T-s)U}(W)) + g_2(X^{N,s,y}_T(W))\right].$$

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

$$\psi^N(s,y,g_1,g_2,W,U) = (T-s)g_1(s+(T-s)U, X^{N,s,y}_{s+(T-s)U}(W)) + g_2(X^{N,s,y}_T(W)) \qquad (3.2)$$

such that $\Psi^N(s,y,g_1,g_2,W) = \mathbb{E}_U[\psi^N(s,y,g_1,g_2,W,U)]$.

From a practical point of view, the PDE (2.3) is solved on $[0,T]\times\mathcal{D}$ where $\mathcal{D}\subset\mathbb{R}^d$ such that $\sup_{0\le t\le T}|X_t|\in\mathcal{D}$ with a probability 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 $k-1$. 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 $(u^k(t_i^k,x_i^k))_{1\le i\le n}$. Now, we know the vector $(u^k+c^k)(t_i^k,x_i^k))_{1\le i\le n}$. From these values, we extrapolate the function $u^{k+1}=u^k+c^k$ on $[0,T]\times\mathcal{D}$.*

$$u^{k+1}(t,x) = \mathcal{P}^k(u^k+c^k)(t,x), \quad \text{for } (t,x)\in[0,T]\times\mathcal{D}, \qquad (3.3)$$

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.

Since c^k is computed using Monte-Carlo simulations instead of a true expectation, the values $(c^k(t_i^k,x_i^k))_{1\le i\le n}$ are random variables. Therefore, u^{k+1} is a random function depending on the random variables needed to compute u_k and $W^{m,k,i}, U^{m,k,i}$, $1\le m\le M, 1\le i\le n$. In view of this comment, the σ-algebra $\mathcal{G}^k$ has to be redefined to take into account this new source of randomness.

Definition 2 (Definition of the σ-algebra $\mathcal{G}^k$)**.** *Let $\mathcal{G}^{k+1}$ define the σ-algebra generated by the set of all random variables used to build u^{k+1}. Using (3.3) yields*

$$\mathcal{G}^{k+1} = \mathcal{G}^k \vee \sigma(\mathcal{A}^k, \mathcal{S}^k),$$

where $\mathcal{A}^k$ is the set of random points used at step k to build the estimator $\mathcal{P}^k$ (see below), $\mathcal{S}^k = \{W^{m,k,i}, U^{m,k,i}, 1 \le m \le M, 1 \le i \le n\}$, is the set of independent Brownian motions used to simulate the paths $X^{m,k,N}(x_i^k)$, and $\mathcal{G}^k$ is the σ-algebra generated by the set of all random variables used to build u^k.

3.2 Choice of the operator

The most delicate part of the algorithm is how to extrapolate a function h and its derivatives when only knowing its values at n points $(t_i, x_i)_{i=1,\dots,n} \in [0,T] \times \mathcal{D}$.

3.2.1 A kernel operator

In the first version of Algorithm 1 presented in Gobet and Labart (2010), a function h was extrapolated from the values computed on the grid by using a kernel operator of the form

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

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) requires $O(n \times d)$ computations.

The convergence result established by Gobet and Labart (2010, Theorem 5.1) is 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 is quite hard to tune for each new problem. Hence, we have tried to use an other solution. Basically, we have borrowed the solution proposed by Longstaff and Schwartz (2001) 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

$$\boldsymbol{\alpha} = \arg\min_{\alpha \in \mathbb{R}^p} \sum_{i=1}^{n} \left| y_i - \sum_{l=1}^{p} \alpha_l B_l(t_i, x_i) \right|^2, \tag{3.4}$$

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.

Practical determination of the vector $\boldsymbol{\alpha}$ In this part, we use the notation $d' = d + 1$. It is quite easy to see from Equation (3.4) 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 (3.4) and the vector $\boldsymbol{\alpha}$ solves

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

$$\boldsymbol{A}\boldsymbol{\alpha} = \sum_{i=1}^{n} y_i B(t_i, x_i) \tag{3.5}$$

where the $p \times p$ matrix $\boldsymbol{A} = (\sum_{i=1}^{n} B_l(t_i, x_i) B_j(t_i, x_i))_{l,j=1,\ldots,p}$ and the vector $B = (B_1, \ldots, B_p)^*$. 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 $\boldsymbol{A}$ is not fate; we can improve the situation 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 $O(p^2 n d')$. 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 $O(p^3 + np^2 d')$.

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 $\text{Vect}\{B_l,\ l = 1, \ldots, p\}$ 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 2 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 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 $q \leq 1$

$$\left\{ \nu \in \mathbb{N}^{d'} : \left(\sum_{i=1}^{d'} \nu_i^q \right)^{1/q} \leq \eta \right\}. \tag{3.6}$$

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.

4 Parallel approach

In this part, we present a parallel version of Algorithm 1, 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 solved them.

4.1 Detailed presentation of the algorithm

Here are the notations we use in the algorithm.

- $\boldsymbol{u}^k = (u^k(t_i^k, x_i^k))_{1 \leq i \leq n} \quad \in \mathbb{R}^n$
- $\boldsymbol{c}^k = (c^k(t_i^k, x_i^k))_{1 \leq i \leq n} \quad \in \mathbb{R}^n$
- n: number of points of the grid
- 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 Iterative algorithm

```
1:  $u^0 \equiv 0$, $\boldsymbol{\alpha}^0 \equiv 0$.
2:  for $k = 0 : K_{it} - 1$ do
3:      Pick at random $n$ points $(t_i^k, x_i^k)_{1 \le i \le n}$.
4:      for $i = 1 : n$ do
5:          for $m = 1 : M$ do
6:              Let $\boldsymbol{W}$ be a Brownian motion with values in $\mathbb{R}^d$ discretized
                  on a time grid with $N$ time steps.
7:              Let $U \sim \mathcal{U}_{[0,1]}$.
8:              Compute
                  $$a_m^{i,k} = \psi^N\left(t_i^k, x_i^k, f_{u^k} + (\partial_t + \mathcal{L}^N)u^k, \Phi - u^k, \boldsymbol{W}, U\right).$$
                /* We recall that $u^k(t,x) = \sum_{l=1}^p \boldsymbol{\alpha}_l^k B_l(t,x)$ */
9:          end for
              $$\boldsymbol{c}_i^k = \frac{1}{M}\sum_{m=1}^M a_m^{i,k} \quad (4.1)$$
              $$\boldsymbol{u}_i^k = \sum_{l=1}^p \boldsymbol{\alpha}_l^k B_l(t_i^k, x_i^k) \quad (4.2)$$
10:     end for
11:     Compute
          $$\boldsymbol{\alpha}^{k+1} = \arg\min_{\alpha \in \mathbb{R}^p} \sum_{i=1}^n \left|(\boldsymbol{u}_i^k + \boldsymbol{c}_i^k) - \sum_{l=1}^p \alpha_l B_l(t_i^k, x_i^k)\right|^2. \quad (4.3)$$
12: 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 a minimum of communications. In that respect, an embarrassingly parallel example is the well-known Monte-Carlo algorithm. Unfortunately, Algorithm 1 is far from being so simple. 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

As we have just explained, we have decided to parallelize the loop over i (line 4 in Algorithm 1). We have used a *Robbin Hood* approach. In the following, we assume to have $P + 1$ processors at hand with $n > P$. We use the following master slave scheme:

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 first P points $(t_i^k, x_i^k)_{1\leq i\leq P}$ to the P slave processors and assign each of them the computation of the corresponding $\boldsymbol{c}_i^k$.
3. As soon as a processor finishes its computation, it sends its result back to the master which in turn sends it a new point (t_i^k, x_i^k) at which evaluating ψ^N to compute $\boldsymbol{c}_i^k$ and this process goes on until all the $(\boldsymbol{c}_i^k)_{i=1,\dots,n}$ have been computed.

At the end of this process, the master knows $\boldsymbol{c}^k$, which corresponds to the approximation of the correction at the random points $(t_i^k, x_i^k)_{1\leq i\leq n}$. From these values and the vector $\boldsymbol{u}^k$, the master computes $\boldsymbol{\alpha}^{k+1}$ and the algorithm can go through a new iteration over k.

What is the best way to send the data to each slave process?

- Before starting any computations, assign to each process a block of iterations over i and send the corresponding data all at once. This way, just one connection has to be initialised which is faster. But the time spent by the master to take care of its slave is longer which implies that at the beginning the slave process will remain longer unemployed. When applying such a strategy, we implicitly assume that all the iterations have the same computational cost.
- Send data iteration by iteration. The latency at the beginning is smaller than in the block strategy and performs better when all iterations do not have the same computational cost.

Considering the wide range of data to be sent and the intensive use of elaborate structures, the most natural way to pass these objects was to rely on the packing mechanism of MPI. Moreover, the library we are using in the code (see Section 4.5) already has a MPI binding which makes the manipulation of the different objects all the more easy. The use of packing enabled to reduce the number of communications between the master process and a slave process to just one communication at the beginning of each iteration of the loop over i (line 4 of Algorithm 1).

4.4 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.

Several strategies exist for that.

1. Newbies in parallel computing might be tempted to take any random number generator and fix a different seed on each processor. Although this naive strategy often leads to satisfactory results on toy examples, it can induce an important bias and lead to detrimental results.
2. The first reasonable approach is to split a sequence of random numbers across several processors. To efficiently implement this strategy, the generator must have 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.

3. 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. The drawback of the DCMT is that its initialization process might be quite lengthy, actually the CPU time needed to create a new generator is not bounded. We give in Figure 1 the distribution.

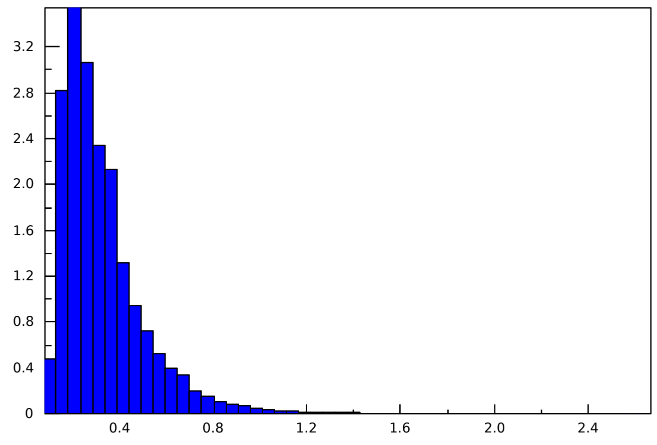

Figure 1: Distribution of the CPU time needed for the creation of one Mersenne Twister generator

4.5 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 implementing mathematics and more recently some MPI bindings have been added to easily manipulate the different objects available in PNL such as vectors and matrices. 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 Pricing and Hedging American options in local volatility models

In this Section, we present a method to price and hedge American options by using Algorithm 1, which solves standard BSDEs.

5.1 Framework

Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space and $(W_t)_{t\geq 0}$ 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 d risky assets, with prices $S_t^1, \cdots, S_t^d$ at time t, and let X_t be the d-dimensional vector of log-returns $X_t^i = \log S_t^i$, for $i = 1, \cdots, d$. We assume that (X_t) satisfies the following stochastic differential equation:

$$dX_t^i = \left(r(t) - \delta_i(t) - \frac{1}{2} \sum_{j=1}^d {\sigma_{ij}}^2(t, e^{X_t}) \right) dt + \sum_{j=1}^d \sigma_{ij}(t, e^{X_t}) dW_t^j, \ i = 1, \cdots, d \qquad (5.1)$$

on a finite interval $[0, T]$, where T is the maturity of the option. We denote by $X_s^{t,x}$ a continuous version of the flow of the stochastic differential Equation (5.1). $X_t^{t,x} = x$ almost surely.

In the following, we assume

Hypothesis 2

1. *$r : [0, T] \longmapsto \mathbb{R}$ is a C_b^1 function. $\delta : [0, T] \longmapsto \mathbb{R}^d$ is a C_b^1 function.*
2. *$\sigma : [0, T] \times \mathbb{R}^d \longmapsto \mathbb{R}^{d\times d}$ is a $C_b^{1,2}$ function.*
3. *σ satisfies the following coercivity property:*

$$\exists \varepsilon > 0 \ \forall (t, x) \in [0, T] \times \mathbb{R}^d, \forall \xi \in \mathbb{R}^d \sum_{1\leq i,j\leq d} [\sigma\sigma^*]_{i,j}(t, x)\xi_i\xi_j \geq \varepsilon \sum_{i=1}^d \xi_i^2$$

We are interested in computing the price of an American option with payoff $\Phi(X_t)$, where $\Phi : \mathbb{R}^d \longmapsto \mathbb{R}_+$ is a continuous function (in case of a put option, $\Phi(x) = (K - \frac{1}{d}(e^{x_1} + \cdots + e^{x_d})_+)$.

From Jaillet et al. (1990), we know that if

Hypothesis 3 Φ *is continuous and satisfies* $|\Phi(x)| \leq Me^{M|x|}$ *for some* $M > 0$*,*

the price at time t of the American option with payoff Φ is given by

$$V(t, X_t) = \text{esssup}_{\tau \in \mathcal{T}_{t,T}} \mathbb{E}\left(e^{-\int_t^\tau r(s)ds} \Phi(X_\tau) | \mathcal{F}_t\right),$$

where $\mathcal{T}_{t,T}$ is the set of all stopping times with values in $[t, T]$ and $V : [0, T] \times \mathbb{R}^d \longmapsto \mathbb{R}_+$ defined by $V(t, x) = \sup_{\tau \in \mathcal{T}_{t,T}} \mathbb{E}\left(e^{-\int_t^\tau r(s)ds} \Phi(X_\tau^{t,x})\right)$.
We also introduce the amount $\Delta(t, X_t)$ involved in the asset at time t to hedge the American option. The function Δ is given by $\Delta(t, x) = \nabla_x V(t, x)$.

In order to link American option prices to BSDEs, we need three steps:

1. writing the price of an American option as a solution of a variational inequality (see Section 5.2)
2. linking the solution of a variational inequality to the solution of a reflected BSDE (see Section 5.3)
3. approximating the solution of a RBSDE by a sequence of standard BSDEs (see Section 5.4)

We refer to Jaillet et al. (1990) for the first step and to El Karoui et al. (1997a) for the second and third steps.

5.2 American option and Variational Inequality

First, we recall the variational inequality satisfied by V. We refer to Jaillet et al. (1990, Theorem 3.1) for more details. Under Hypotheses 2 and 3, V solves the following parabolic PDE

$$\begin{cases} \max(\Phi(x) - u(t, x), \partial_t u(t, x) + \mathcal{A}u(t, x) - r(t)u(t, x)) = 0, \\ u(T, x) = \Phi(x). \end{cases} \tag{5.2}$$

where
$\mathcal{A}u(t, x) = \sum_{i=1}^d (r(t) - \delta_i(t) - \frac{1}{2} \sum_{j=1}^d \sigma_{ij}^2(t, e^x)) \partial_{x_i} u(t, x) + \frac{1}{2} \sum_{1 \leq i,j \leq d} [\sigma\sigma^*]_{ij}(t, e^x) \partial^2_{x_i x_j} u(t, x)$ is the generator of X.

5.3 Variational Inequality and Reflected BSDEs

This section is based on El Karoui et al. (1997a, Theorem 8.5). We assume

Hypothesis 4 Φ *is continuous and has at most a polynomial growth (ie,* $\exists C, p > 0$ *s.t.* $|\Phi(x)| \leq C(1 + |x|^p)$*).*

Let us consider the following reflected BSDE

$$\begin{cases} -dY_t = -r(t)Y_t dt - Z_t dW_t + dH_t, \\ Y_T = \Phi(X_T),\ Y_t \geq \Phi(X_t), \forall t, \\ \int_0^T (Y_t - \Phi(X_t)) dH_t = 0. \end{cases} \tag{5.3}$$

Then, under Hypotheses 2 and 4, $u(t,x) = Y_t^{t,x}$ is a viscosity solution of the obstacle problem (5.2), where $Y_t^{t,x}$ is the value at time t of the solution of (5.3) on $[t,T]$ where the superscripts t,x mean that X starts from x at time t. We also have $(\nabla_x u(t,x))^* \sigma(t,x) = Z_t^{t,x}$ (* means transpose). Then, we get that the price V and the delta Δ of the option are given by

$$V(t,X_t) = Y_t, \ \Delta(t,X_t) = (Z_t \sigma(t,X_t)^{-1})^*.$$

5.4 Approximation of a RBSDE by a sequence of standard BSDEs

We present a way of approximating a RBSDE by a sequence of standard BSDEs. The idea was introduced by El Karoui et al. (1997a, Section 6) for proving the existence of a solution to RBSDE by turning the constraint $Y_t \geq \Phi(X_t)$ into a penalisation. Let us consider the following sequence of BSDEs indexed by i

$$Y_t^i = \Phi(X_T) - \int_t^T r(s) Y_s^i ds + i \int_t^T (Y_s^i - \Phi(X_s))^- ds - \int_t^T Z_s^i dW_s, \tag{5.4}$$

whose solutions are denoted (Y^i, Z^i). We define $H_t^i = i \int_t^T (Y_s^i - \Phi(X_s))^- ds$. Under Hypotheses 2 and 4, the sequence $(Y^i, Z^i, H^i)_i$ converges to the solution (Y,Z,H) of the RBSDE (5.3), when i goes to infinity. Moreover, Y^i converges increasingly to Y. The term H^i is often called a penalisation. From a practical point, there is no use solving such a sequence of BSDEs, because we can directly apply our algorithm to solve Equation (5.4) for a given i. Therefore, we actually consider the following penalized BSDE

$$Y_t = \Phi(X_T) - \int_t^T r(s) Y_s ds + \omega \int_t^T (Y_s - \Phi(X_s))^- ds - \int_t^T Z_s dW_s, \tag{5.5}$$

where $\omega \geq 0$ is penalization weight. In practice, the magnitude of ω must remain reasonably small as it appears as a contraction constant when studying the speed of convergence of our algorithm. Hence, the larger ω is, the slower our algorithm converges. So, a trade-off has to be found between the convergence of our algorithm to the solution Y of (5.5) and the accuracy of the approximation of the American option price by Y.

5.5 European options

Let us consider a European option with payoff $\Phi(X_T)$, where X follows (5.1). We denote by V the option price and by Δ the hedging strategy associated to the option. From El Karoui et al. (1997b), we know that the couple (V, Δ) satisfies

$$-dV_t = -r(t) V_t dt - \Delta_t^* \sigma(t, X_t) dW_t, V_T = \Phi(X_T).$$

Then, (V, Δ) is solution of a standard BSDE. This corresponds to the particular case $\omega = 0$ of (5.5), Y corresponding to V and Z to $\Delta_t^* \sigma(t, X_t)$.

6 Numerical results and performance

6.1 The cluster

All our performance tests have been carried out on a $256-$PC cluster from SUPELEC Metz. Each node is a dual core processor : INTEL Xeon-3075 2.66 GHz with a front side bus at

1333Mhz. The two cores of each node share 4GB of RAM and all the nodes are interconnected using a Gigabit Ethernet network. In none of the experiments, did we make the most of the dual core architecture since our code is one threaded. Hence, in our implementation a dual core processor is actually seen as two single core processors.

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

6.2 Black-Scholes' framework

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

$$dS^i_t = S^i_t((r-\delta_i)dt + \sigma^i dW^i_t) \qquad S_0 = (S^1_0, \dots, S^d_0) \tag{6.1}$$

where $W = (W^1, \dots, W^d)$. Each component W^i is a standard Brownian motion. For the numerical experiments, the covariance structure of W will be assumed to be given by $\langle W^i, W^j \rangle_t = \rho t \mathbf{1}_{\{i \neq j\}} + t\mathbf{1}_{\{i=j\}}$. 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 $C = LL^*$. To simulate W on the time-grid $0 < t_1 < t_2 < \ldots < t_N$, we need $d \times N$ independent standard normal variables and set

$$\begin{pmatrix} W_{t_1} \\ W_{t_2} \\ \vdots \\ W_{t_{N-1}} \\ W_{t_N} \end{pmatrix} = \begin{pmatrix} \sqrt{t_1}L & 0 & 0 & \dots & 0 \\ \sqrt{t_1}L & \sqrt{t_2-t_1}L & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \sqrt{t_{N-1}-t_{N-2}}L & 0 \\ \sqrt{t_1}L & \sqrt{t_2-t_1}L & \dots & \sqrt{t_{N-1}-t_{N-2}}L & \sqrt{t_N-t_{N-1}}L \end{pmatrix} G,$$

where G is a normal random vector in $\mathbb{R}^{d\times N}$. The vector $\sigma = (\sigma^1, \dots, \sigma^d)$ is the vector of volatilities, $\delta = (\delta^1, \dots, \delta^d)$ is the vector of instantaneous dividend rates and $r > 0$ is the instantaneous interest rate. We will denote the maturity time by T. Since we know how to simulate the law of (S_t, S_T) exactly for $t < T$, there is no use to discretize equation (6.1) using the Euler scheme. In this Section $N = 2$.

6.2.1 European options

We want to study the numerical accuracy of our algorithm and to do that we first 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, while it is no more the case for American options.
In this paragraph, the parameter ω appearing in (5.5) is 0.

European put basket option. Consider the following put basket option with maturity T

$$\left(K - \frac{1}{d}\sum_{i=1}^{d} S^i_T\right)_+ \tag{6.2}$$

Figure 2 presents the influence of the parameters M, n and q. The results obtained for $n = 1000$, $M = 50,000$ and $q = 1$ (curve (+)) are very close to the true price and moreover we can see that the algorithm stabilizes after very few iterations (less than 10).

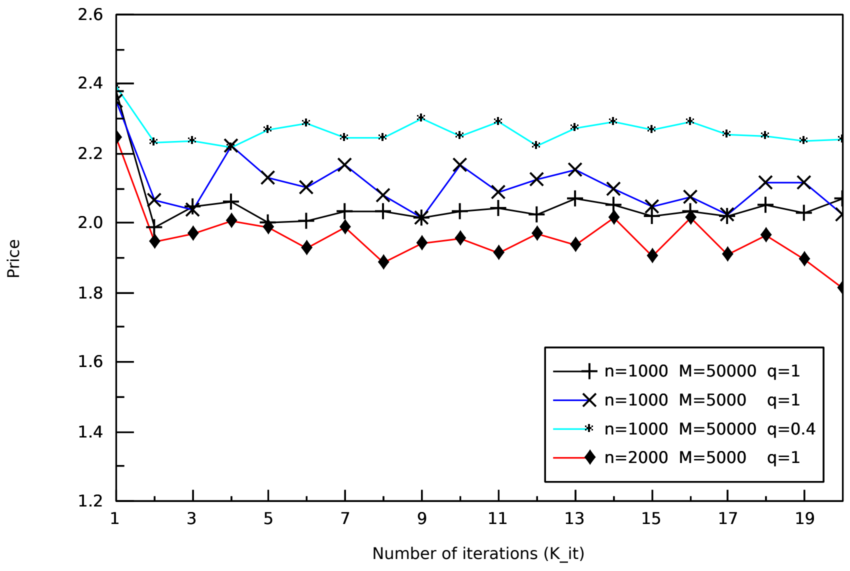

Figure 2: Convergence of the algorithm for a European put basket option with $d = 5$, $\rho = 0.1$, $T = 3$, $S_0 = 100$, $\sigma = 0.2$, $\delta = 0$, $r = 0.05$ $K = 100$, $\eta = 3$, $\omega = 0$. The benchmark price computed with a high precision Monte–Carlo method yields 2.0353 with a confidence interval of $(2.0323, 2.0383)$.

- Influence of M: curves $(+)$ and $(\times)$ show that taking $M = 5000$ is not enough to get the stabilization of the algorithm.
- Joined influence of n and M: curves $(\times)$ and $(\blacklozenge)$ show the importance of well balancing the number of discretization points n with the number of Monte–Carlo simulations M. The sharp behaviour of the curve $(\blacklozenge)$ may look surprising at first, since we are tempted to think that a larger numbers of points n will increase the accuracy. However, increasing the number of points but keeping the number of Monte–Carlo simulations constant creates an over fitting phenomenon because the Monte–Carlo errors arising at each point are too large and independent and it leads the approximation astray.
- Influence of q: we can see on curves $(+)$ and $(*)$ that decreasing the hyperbolic index q can lead a highly biased although smooth convergence. This highlights the impact of the choice of q on the solution computed by the algorithm.

To conclude, we notice that the larger the number of Monte–Carlo simulations is, the smoother the convergence is, but when the polynomial family considered is too sparse it can lead to a biased convergence.

European call basket option. Consider the following put basket option with maturity T

$$\left(\frac{1}{d}\sum_{i=1}^{d} S_T^i - K\right)_+ \tag{6.3}$$

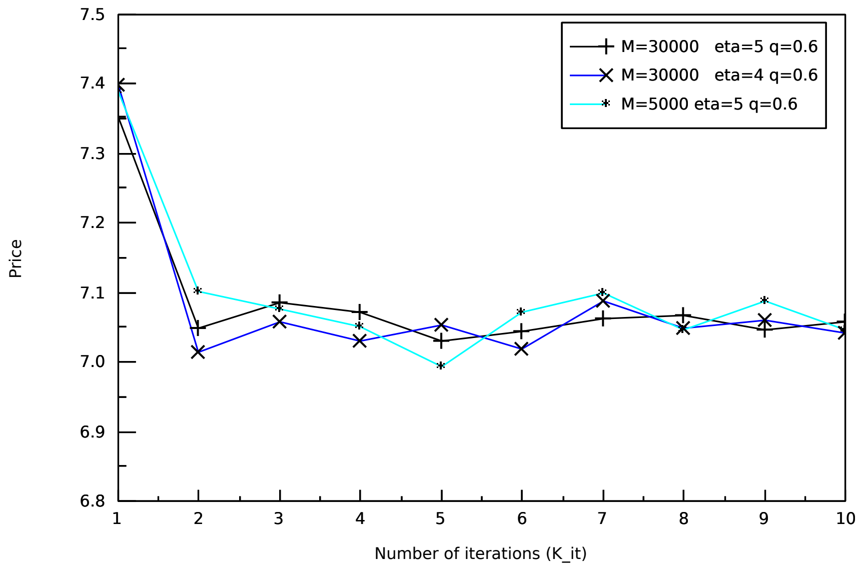

Figure 3: Convergence of the price of a European call basket option with $d = 10$, $\rho = 0.2$, $T = 1$, $S_0 = 100$, $\sigma = 0.2$, $\delta = 0$, $r = 0.05$, $K = 100$, $n = 1000$ and $\omega = 0$. The benchmark price computed with a high precision Monte–Carlo method yields 7.0207 with a confidence interval of $(7.0125, 7.0288)$.

Figure 3 illustrates the impact of the sparsity of the polynomial basis considered on the convergence of the algorithm. The smoothest convergence is achieved by the curve $(+)$, ie when $M = 30,000$, $\eta = 5$ and $q = 0.6$. The algorithm stabilizes very close to the true price and after very few iterations.

- Influence of η : for a fixed value of q, the sparsity increases when η decreases, so the basis with $\eta = 4, q = 0.6$ is more sparse than the one with $\eta = 5, q = 0.6$. We compare curves $(+)$ $(\eta = 5)$ and $(\times)$ $(\eta = 4)$ for fixed values of q $(= 0.6)$ and M $(= 30,000)$. We can see that for $\eta = 4$ (curve $(\times)$) the algorithm stabilizes after 7 iterations, whereas for $\eta = 6$ (curve $(+)$) less iterations are needed to converge.

- Influence of M : for fixed values of η $(= 5)$ and q $(= 0.6)$, we compare curves $(+)$ $(M = 30000)$ and $(*)$ $(M = 5000)$. Using a large number of simulations is not enough to get a good convergence, as it is shown by 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 a noise which has a similar effect to reducing the number of Monte–Carlo simulations. This is precisely what we observe on Figure 3: the curves $(\times)$ and $(*)$ have a very similar behaviour although one of them uses a much larger number of simulations.

6.2.2 American options

In this paragraph, the penalisation parameter ω appearing in (5.5) is 1.

Pricing American put basket options. We have tested our algorithm on the pricing of a multidimensional American put option with payoff given by Equation (6.2)

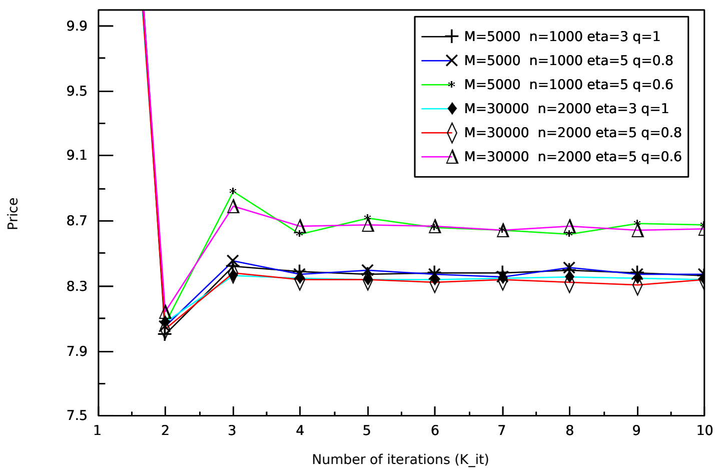

Figure 4: Convergence of the price of an American put basket option with $d = 5$, $\rho = 0.2$, $T = 1$, $S_0 = 100$, $\sigma = 0.25$, $\delta = 0.1$, $K = 100$, $r = 0.05$ and $\omega = 1$.

Figure 4 presents the influence of the parameters M and q.

- Influence of M : when zooming on Figure 4, one can indeed see that the curves using 30000 Monte–Carlo simulations are a little smoother than the others but these extra simulations do not improve the convergence as much as in Figures 2 and 3 (compare curves (+) and (♦), Figure 4). The main explanation of this fact is that put options have in general less variance than call options and in Figure 2 a maturity of $T = 3$ was used which leads to a larger variance than with $T = 1$.
- Influence of q : once again, we can observe that increasing the sparsity of the polynomial basis (ie, decreasing q) can lead to a biased convergence. When $q = 0.6$, we get a biased result (see curves ($*$) and ($\triangle$)), even for M large (curve ($\triangle$), $M = 30000$).

Then, it is advisable for American put options to use almost full polynomial basis with fewer Monte–Carlo simulations in order to master the computational cost rather than doing the contrary.

Hedging American put basket options. Now, let us present the convergence of the approximation of the delta at time 0. Table 1 presents the values of the delta of an American put basket option when the iterations increase. We see that the convergence is very fast (we only need 3 iterations to get a stabilized value). The parameters of the algorithm are the following ones: $n = 1000$, $M = 5000$, $q = 1$, $\eta = 3$ and $\omega = 1$.

Iteration	Δ^1	Δ^2	Δ^3	Δ^4	Δ^5
1	-0.203931	-0.205921	-0.203091	-0.205264	-0.201944
2	-0.105780	-0.102066	-0.103164	-0.102849	-0.108371
3	-0.109047	-0.105929	-0.105604	-0.105520	-0.111327
4	-0.108905	-0.105687	-0.105841	-0.105774	-0.111137
5	-0.108961	-0.105648	-0.105725	-0.105647	-0.111274

Table 1: Convergence of the delta for an American put basket option with $d = 5$, $\rho = 0.2$, $T = 1$, $S_0 = 100$, $\sigma = 0.25$, $\delta = 0.1$, $K = 100$, $r = 0.05$.

6.3 Dupire's framework

We consider a d-dimensional local volatility model in which the dynamics under the risk-neutral measure of each asset is supposed to be given by

$$dS_t^i = S_t^i((r-\delta_i)dt + \sigma(t, S_t^i)dW_t^i) \qquad S_0 = (S_0^1, \ldots, S_0^d)$$

where $W = (W^1, \ldots, W^d)$ is defined and generated as in the Black-Scholes framework. The local volatility function σ we have chosen is of the form

$$\sigma(t,x) = 0.6(1.2 - e^{-0.1t}e^{-0.001(xe^{rt}-s)^2})e^{-0.05\sqrt{t}}, \tag{6.4}$$

with $s > 0$. Since there exists a duality between the variables (t, x) and (T, K) in Dupire's framework, one should choose s equal to the spot price of the underlying asset. Then, the bottom of the smile is located at the forward money. The parameters of the algorithm in this paragraph are the following : $n = 1000$, $M = 30000$, $N = 10$, $q = 1$, $\eta = 3$.

Pricing and Hedging European put basket options. We consider the put basket option with payoff given by (6.2). The benchmark price and delta are computed using the algorithm proposed by Jourdain and Lelong (2009), which is based on Monte-Carlo methods.

Concerning the delta, we get at the last iteration the following vector $\Delta = (-0.062403 - 0.061271 - 0.062437 - 0.069120 - 0.064743)$. The benchmark delta is -0.0625.

Pricing and Hedging American put basket options. We still consider the put payoff given by (6.2). In the case of American options in local volatility models, there is no benchmark. However, Figure 6 shows that the algorithm converges after few iterations. We get a price around 6.30. At the last iteration, we get $\Delta = (-0.102159 - 0.102893 - 0.103237 - 0.110546 - 0.106442)$.

6.4 Speed up.

Remark 3. *Because in high dimensions, the sequential algorithm can run several hours before giving a price, we could not afford to run the sequential algorithm on the cluster to have a benchmark value for the reference CPU time used in the speed up measurements. Instead we have computed speed ups as the ratio*

$$\textit{speed up} = \frac{\textit{CPU time for } 8 \textit{ processors / } 8}{\textit{CPU time for } n \textit{ processors} \times n} \tag{6.5}$$

This explains why we may get in the tables below some speed ups slightly greater than 1.

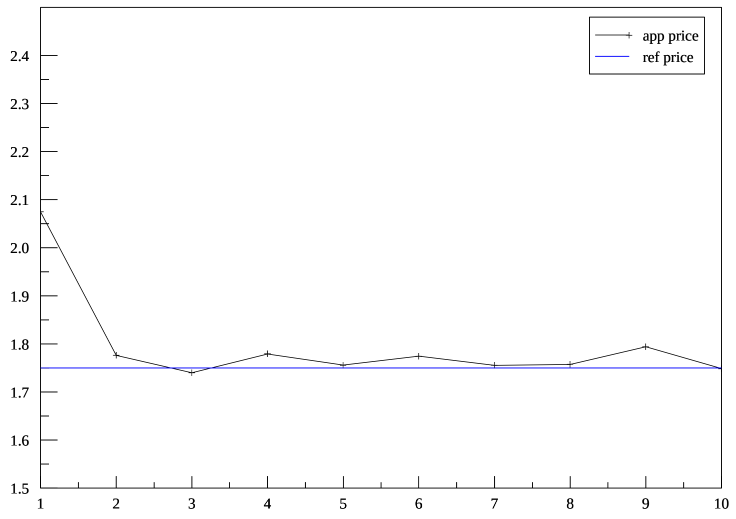

Figure 5: Convergence of the algorithm for a European put basket option with $d = 5$, $\rho = 0$, $T = 1$, $S_0 = 100$, $\delta = 0$, $K = 100$, $r = 0.05$, $\eta = 3$, $\omega = 0$. The benchmark price computed with a high precision Monte–Carlo method yields 1.745899 with a confidence interval of $(1.737899, 1.753899)$.

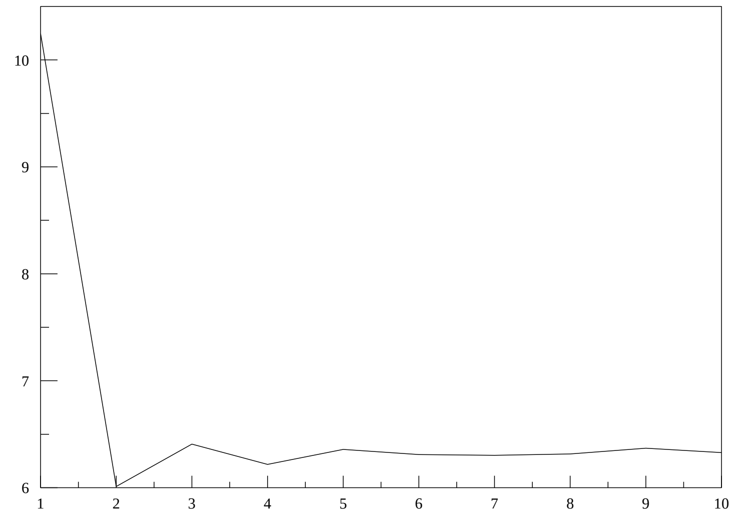

Figure 6: Convergence of the algorithm for an American put basket option with $d = 5$, $\rho = 0$, $T = 1$, $S_0 = 100$, $\delta = 0.1$, $K = 100$, $r = 0.05$ and $\omega = 1$.

Our goal in this paper was to design a scalable algorithm for high dimensional problems, so it is not surprising that the algorithm does not behave so well in relatively small dimension as highlighted in Table 2. In dimension 3, the speed ups are linear up to 28 processors but then they dramatically decrease toward zero: this can be explained by the small CPU load of the computation of the correction term at a given point (t_i, x_i). The cost of each iteration of the loop line 4 of Algorithm 1 is proportional to Mpd^3 and when d is small so is p — the number of polynomials of total degree less or equal than η. For instance, for $d = 3$ and $\eta = 3$, we have $p = 20$, which gives a small complexity for each iteration over i. Hence, when the number of processors used increases, the amount of work to be done by its processor between two synchronisation points decreases to such a point that most of the CPU time is used for transmitting data or waiting. This explains why the speed ups decrease so much. Actually, we were expecting such results as the parallel implementation of the algorithm has been designed for high dimensional problems in which the amount of work to be done by each processor cannot decrease so much unless several dozens of thousands of processors are used. This phenomena can be observed in Table 3 which shows very impressive speed ups: in dimension 6, even with 256 processors the speed ups are still linear which highlights the scalability of our implementation. Even though computational times may look a little higher than with other algorithms, one should keep in mind that our algorithm not only computes prices but also hedges, therefore the efficiency of the algorithm remains quite impressive.

Nb proc.	Time	Speed up
8	543.677	1
16	262.047	1.03737
18	233.082	1.03669
20	210.114	1.03501
24	177.235	1.02252
28	158.311	0.981206
32	140.858	0.964936
64	97.0629	0.70016
128	103.513	0.328267
256	162.936	0.104274

Table 2: Speed ups for the American put option with $d = 3$, $r = 0.02$, $T = 1$, $\sigma = 0.2$, $\rho = 0$, $S_0 = 100$, $K = 95$, $M = 1000$, $N = 10$, $K = 10$, $n = 2000$, $r = 3$, $q = 1$, $\omega = 1$. See Equation (6.5) for the definition of the "Speed up" column.

7 Conclusion

In this work, we have presented a parallel algorithm for solving BSDE and applied it to the pricing and hedging of American option which remains a computationally demanding problem for which very few scalable implementations have been studied. Our parallel algorithm shows an impressive 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.

Nb proc.	Time	Speed up
8	1196.79	1
16	562.888	1.06308
24	367.007	1.08698
32	272.403	1.09836
40	217.451	1.10075
48	181.263	1.10042
56	154.785	1.10457
64	135.979	1.10016
72	121.602	1.09354
80	109.217	1.09579
88	99.6925	1.09135
96	91.9594	1.08453
102	85.6052	1.0965
110	80.2032	1.08523
116	75.9477	1.08676
128	68.6815	1.08908
256	35.9239	1.04108

Table 3: Speed ups for the American put option with $d = 6$, $r = 0.02$, $T = 1$, $\sigma = 0.2$, $\rho = 0$, $S_0 = 100$, $K = 95$, $M = 5000$, $N = 10$, $K = 10$, $n = 2000$, $r = 3$, $q = 1$, $\omega = 1$. See Equation (6.5) for the definition of the “Speed up“ column.

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.

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.

V. Dung Doan, A. Gaiwad, M. Bossy, F. Baude, and I. Stokes-Rees. Parallel pricing algorithms for multimensional bermudan/american options using Monte Carlo methods. *Mathematics and Computers in Simulation*, 81(3):568–577, 2010.

N. El Karoui, C. Kapoudjian, E. Pardoux, S. Peng, and M. Quenez. Reflected solutions of backward SDE's, and related obstacle problems for PDE's. *the Annals of Probability*, 25 (2):702–737, 1997a.

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

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.

K. Huang and R. Thularisam. Parallel Algorithm for pricing american Asian Options with Multi-Dimensional Assets. *Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 177–185, 2005.

A. Ibáñez and F. Zapatero. Valuation by simulation of american options through computation of the optimal exercise frontier. *Journal of Financial Quantitative Analysis*, 93:253–275, 2004.

P. Jaillet, D. Lamberton, and B. Lapeyre. Variational inequalities and the pricing of American options. *Acta Appl. Math.*, 21:263–289, 1990.

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.

F. Longstaff and R. Schwartz. Valuing American options by simulation : A simple least-square approach. *Review of Financial Studies*, 14:113–147, 2001.

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`.

G. Pagès and B. Wilbertz. GPGPUs in computational finance: Massive parallel computing for American style options. *ArXiv e-prints*, Jan. 2011. URL `http://arxiv.org/abs/1101.3228v1`.

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

R. K. Thulasiram and D. A. Bondarenko. Performance evaluation of parallel algorithms for pricing multidimensional. In *ICPP Workshops*, pages 306–313, 2002.

I. Toke and J. Girard. Monte Carlo Valuation of Multidimensional American Options Through Grid Computing. In *Lecture notes in computer science*, volume 3743, pages 462–469. Springer-Verlag, 2006.

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