Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

Domain decomposition solution of nonlinear two-dimensional parabolic problems by random trees

Juan A. Acebrón [a,*], Ángel Rodríguez-Rozas [a], Renato Spigler [b]

[a] Center for Mathematics and its Applications, Department of Mathematics, Instituto Superior Técnico Av. Rovisco Pais 1049-001 Lisboa, Portugal
[b] Dipartimento di Matematica, Università "Roma Tre", Largo S.L. Murialdo 1, 00146 Rome, Italy

ARTICLE INFO

Article history:
Received 29 January 2009
Received in revised form 22 April 2009
Accepted 23 April 2009
Available online 6 May 2009

PACS:
65C05
65C30
65N55

Keywords:
Domain decomposition
Monte Carlo methods
Nonlinear parabolic problems
Random trees
Parallel computing
Fault tolerant algorithms

ABSTRACT

A domain decomposition method is developed for the numerical solution of nonlinear parabolic partial differential equations in any space dimension, based on the probabilistic representation of solutions as an average of suitable multiplicative functionals. Such a direct probabilistic representation requires generating a number of random trees, whose role is that of the realizations of stochastic processes used in the linear problems. First, only few values of the sought solution inside the space-time domain are computed (by a Monte Carlo method on the trees). An interpolation is then carried out, in order to approximate interfacial values of the solution inside the domain. Thus, a fully decoupled set of sub-problems is obtained. The algorithm is suited to massively parallel implementation, enjoying arbitrary scalability and fault tolerance properties. Pruning the trees is shown to increase appreciably the efficiency of the algorithm. Numerical examples conducted in 2D, including some for the KPP equation, are given.

1. Introduction

Probabilistic methods, based on Monte Carlo simulations, are not used very frequently in low dimension, because of their poor performance. They could be used, however, to obtain the solution to boundary value problems for certain partial differential equations (PDEs) at *few* points internal to the domain. This is possible, in contrast with other more traditional (deterministic) methods, which require solving globally the PDE problem on the entire domain. They can also be used in case the domain is complex, and/or in high dimensions.

In [1,2], linear two-dimensional boundary value problems for elliptic PDEs, were solved exploiting the probabilistic representation of solutions. This approach was coupled to a *domain decomposition* (DD) strategy [15,29], consisting of first obtaining only very few values of the solution, at some points internal to the domain, and then interpolating on such points, thus obtaining continuous approximations of the sought solution on suitable interfaces. At that stage, full decoupling in arbitrarily many subdomains is possible, since interfacial values can be used as boundary values on the boundaries of the various subdomains. This fact represents a definitely more advantageous circumstance, compared to what happens in any other existing deterministic domain decomposition methods. We called our method "probabilistic domain decomposition method" (PDD method, for short).

* Corresponding author.
E-mail addresses: juan.acebron@ist.utl.pt (J.A. Acebrón), angel.rodriguez@ist.utl.pt (Á. Rodríguez-Rozas), spigler@mat.uniroma3.it (R. Spigler).

0021-9991/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2009.04.034

In [5], we extended such method to treat nonlinear parabolic one-dimensional problems. In this paper, we generalize these results to deal with arbitrary space dimensions. For *linear* parabolic problems, a probabilistic representation is known to exist (the Feynman–Kac formula), as it happens for linear elliptic problems. Probabilistic representations for the solution to certain *nonlinear* PDE problems, generally speaking, are less known. A key feature of some of such representations is that they require generating suitable "random trees", whose "realizations" (or "paths") can be thought of as the generalization of the paths of stochastic processes on which averages are computed in the linear cases.

We should recall here the seminal work by McKean [27], where a probabilistic representation was derived for the solution to the nonlinear parabolic equation known as the Kolmogorov–Petrovskii–Piskunov equation (KPP, for short). Other important achievements concerning probabilistic representations of solutions to nonlinear PDEs appeared in [25] for the Fourier transformed Navier–Stokes equations. Subsequent theoretical advances have been accomplished recently, aiming to generalize the results of [25] to other nonlinear PDEs of interest to Physics and Engineering [36]. In [18,34], a theoretical investigation on kinetic equations of the Vlasov-Poisson type has been accomplished and preliminary numerical simulations conducted, showing the computational feasibility of this approach.

It is worth pointing out that in more general problems, the probabilistic description in terms of random trees may be no longer valid, and it is necessary to resort to more complex stochastic processes. Indeed, a probabilistic representation of the solution to the Navier–Stokes equations in $\mathbf{R}^3$ has been derived by means of tree-indexed random trees [28].

Finally, in [30], a linear one-dimensional diffusion equation as well as the one-dimensional viscous Burger's equation have been solved numerically by means of convenient random walks and branching random walks, respectively. The results were then improved by a Picard iteration scheme.

While all of this can be exploited to obtain the solution at some points, internal to the space-time domain, we emphasize the possibility of using parallel computers in a very efficient way. This can be done, as in [1,2], i.e., using the few computed values only as nodes on which one can interpolate, thus obtaining approximations for the interfacial values, and then fully decoupling into independent sub-problems. As an aside, the method is capable of exploiting massively parallel architectures. Indeed, not only the Monte Carlo approach can be trivially but effectively parallelized, but also the so-obtained decomposition in subdomains can exploit parallel architectures. Moreover, the intermediate step of interpolation can also be carried out independently on each interface. Hence, there are three sources of parallelism, so to say, though these three stages are separated sequentially. Ultimately, parallel codes with extremely low communication overhead among processors can be developed. In closing we stress that this method is characterized by some of the most desirable properties for effectively exploiting high-performance supercomputers [4], equipped with hundreds of thousands or millions of processors, such as arbitrary *scalability* and *fault tolerance* [3,19].

The three sources of parallelization are: (1) the Monte Carlo generation of internal node functional values (even each single tree can be runned on an independent processor), (2) the interpolation part (the interpolation on each interface can be accomplished independently), and (3) the domain decomposition solution (that can be assigned to independent local solver, using as many processors). Moreover, each of such three stages enjoys a natural fault tolerant property: (1) if a few processors fail in the Monte Carlo simulations, it will be enough to ignore the results from them. Hence, at a price of a small additional errors, the algorithm will still provide meaningful results. (2) Failure of processors computing interpolating values of the solution on some interfaces may only imply to neglect, temporarily, the solution on those subdomains having such interfaces as part of their boundary. (3) Failure of processors responsible for the numerical solution on some subdomains, finally, can also be temporarily neglected, while the solution computed by the local solvers on the remaining subdomains will be computed correctly. Note that on the interfaces and on the subdomains where the processors failed, the solution can be computed running again the algorithm.

Here is the plan of the paper. In Section 2, some general mathematical preliminaries are recalled, while in Section 3 the algorithm is described. The complexity inherent to the probabilistic part of the algorithm as well as various sources of numerical errors are discussed there. In Section 4, a strategy to improve the computational complexity of the algorithm by pruning the random trees is investigated. Numerical examples for nonlinear two-dimensional PDEs, among which the KPP equation, are given in Section 5. The performance of the PDD method is also assessed comparing our results with those obtained by competitive parallel numerical codes, broadly used in the high-performance scientific community. In a short section at the end, the high points of the paper are summarized.

2. Mathematical preliminaries

A variety of phenomena pertaining to Engineering, Physics, and other Sciences, are governed by diffusion equations. The relations between macroscopic diffusion and the mean statistical effect of the microscopic random (Brownian) motion of molecules goes back, among the others, to Einstein and Smoluchowski. A connection between "stochastic differential equations", that can be thought as ordinary differential equations driven by a certain kind of random noise (Langevin equations), and partial differential equations, was established. Inspired by Feynman's "path integrals" in quantum physics, Kac realized that a similar formulation could be applied to obtain a representation of the solution to the heat equation and to other diffusive (parabolic) linear partial differential equations. This lead to the so-called Feynman–Kac formula. Let $u(\mathbf{x}, t)$ be a bounded function satisfying the Cauchy problem for the linear parabolic partial differential equation,

$$\frac{\partial u}{\partial t} = Lu - c(\mathbf{x},t)u, \quad u(\mathbf{x},0) = f(\mathbf{x}), \tag{1}$$

where $\mathbf{x} \in \mathbf{R}^n$, L is a linear elliptic operator, say $L := a_{ij}(\mathbf{x},t)\partial_i\partial_j + b_i(\mathbf{x},t)\partial_i$ (using the summation convention), with continuous bounded coefficients, $c(\mathbf{x},t) \geqslant 0$ continuous bounded, continuous initial condition, $f(\mathbf{x})$. The probabilistic representation of the solution u to Eq. (1) is given through the Feynman–Kac formula

$$u(\mathbf{x},t) = E\left[f(\boldsymbol{\beta}(t))e^{-\int_0^t c(\boldsymbol{\beta}(s),t-s)ds}\right], \tag{2}$$

see [17,23], e.g. where $\boldsymbol{\beta}(\cdot)$ is the n-dimensional stochastic process starting at $(\mathbf{x},0)$, associated to the operator L, and the expected values are taken with respect to the corresponding measure. When L is the n-dimensional Laplace operator, $\boldsymbol{\beta}(\cdot)$ reduces to the standard n-dimensional Brownian motion, and the measure reduces to the Gaussian measure. In general, the stochastic process $\boldsymbol{\beta}(\cdot)$ is the solution of a system of stochastic differential equations (SDEs) of the Ito type, related to the elliptic operator in (1),

$$d\boldsymbol{\beta} = \mathbf{b}(\mathbf{x},t)dt + \boldsymbol{\sigma}(\mathbf{x},t)d\mathbf{W}(t). \tag{3}$$

Here $\mathbf{W}(t)$ represents the n-dimensional standard Brownian motion (or Wiener process); see [23,10], e.g. for generalities, and [24] for related numerical treatments. As is known, the solution to (3) is a n-dimensional stochastic process, $\boldsymbol{\beta}(t,\omega)$, where ω, usually not indicated explicitly in probability theory, denotes the "chance variable", which ranges on an underlying abstract probability space. The drift vector, $\mathbf{b}$, and the diffusion matrix, $\boldsymbol{\sigma}$, in (3), are related to the coefficients of the elliptic operator in (1) by $\mathbf{b} = (b_1,\ldots,b_n)^T$, and $\boldsymbol{\sigma}\boldsymbol{\sigma}^T = \mathbf{a}$, with $\boldsymbol{\sigma} = \{\sigma_{ij}\}_{i,j=1,\ldots,n}$, $\mathbf{a} = \{a_{ij}\}_{i,j=1,\ldots,n}$.

The representation in Eq. (2) can be generalized to deal with problems on *bounded* domains, say $\Omega \subset \mathbf{R}^n$, where given boundary data $u(\mathbf{x},t)|_{\mathbf{x}\in\partial\Omega} = g(\mathbf{x},t)$ of the Dirichlet type are prescribed. Thus, the following representation holds, for the solution of the problem, being now continuous and bounded on $\overline{\Omega} \times [0,T]$,

$$u(\mathbf{x},t) = E\left[f(\boldsymbol{\beta}(t))e^{-\int_0^t c(\boldsymbol{\beta}(s),t-s)ds}\mathbf{1}_{[\tau_{\partial\Omega}>t]}\right] + E\left[g(\boldsymbol{\beta}(\tau_{\partial\Omega}),t-\tau_{\partial\Omega})e^{-\int_0^{\tau_{\partial\Omega}} c(\boldsymbol{\beta}(s),t-s)ds}\mathbf{1}_{[\tau_{\partial\Omega}<t]}\right]. \tag{4}$$

Here $\tau_{\partial\Omega}$ denotes the first exit (or hitting) time of the path $\boldsymbol{\beta}(\cdot)$, started at $(\mathbf{x},t)$, when $\partial\Omega$ is crossed, and $\mathbf{1}_{[\tau>t]}$ is the characteristic function, which takes the value 1 or 0, depending whether $\tau_{\partial\Omega}$ is or is not greater than t.

The solution to the linear inhomogeneous problem

$$\frac{\partial u}{\partial t} = Lu - c(\mathbf{x},t)u + F(\mathbf{x},t), \tag{5}$$

where $F(\mathbf{x},t)$ is a bounded continuous function of $\mathbf{x}$ and t, can also be represented probabilistically, using the related Green function, which, in turn, can be represented as above, being the solution to the associated homogeneous problem, see [5], e.g.

A probabilistic representation does exist also for nonlinear parabolic equations. In [27], McKean derived the representation

$$u(\mathbf{x},t) = E\left[\prod_{i=1}^{k(\omega)} f(\mathbf{x}_i(t,\omega))\right], \tag{6}$$

for the one-dimensional KPP equation

$$u_t = u_{xx} + u(u-1), \quad -\infty < x < +\infty, \quad t > 0 \tag{7}$$

subject to the initial value $u(x,0) = f(x)$, for $-\infty < x < +\infty$; see also [17]. Here $k(\omega)$ is the random number of branches, and $\mathbf{x}_i(t,\omega)$ is the position of the ith stochastic process surviving at time t. A similar representation, however, can be derived for more general nonlinear as well as higher-dimensional parabolic equation problems, such as

$$\frac{\partial u}{\partial t} = Lu - cu + \sum_{i=2}^{m} \alpha_i u^i, \tag{8}$$

where $m \geqslant 2$ is an integer, $\alpha_i \geqslant 0$, $\sum_{i=2}^m \alpha_i = 1$, and c is a positive constant. Note that the PDEs in (8) include, in particular, KPP equations in any dimensions. With respect to the latter, we can handle also nonlinear PDEs with variable coefficients and drift terms. A representation like that in (6) is based on the generation of *branching* diffusion processes associated to the elliptic operator in Eq. (1) and governed by an exponential random time, S, with probability density $p(S) = c\exp(-cS)$.

In [5], the representation obtained by McKean in [27] was illustrated considering the one-dimensional equation $u_t = Lu - cu + u^2$, with $u(x,0) = f(x)$ prescribed, in the form

$$\begin{aligned} u(x,t) &= E[f(\beta(t))\mathbf{1}_{[S_0>t]}] + \frac{1}{c}E[f(\beta(t-S_0))\mathbf{1}_{[S_1>t-S_0]}f(\beta(t-S_0))\mathbf{1}_{[S_2>t-S_0]}\mathbf{1}_{[S_0<t]} \\ &\quad + \frac{1}{c^2}E[f(\beta(t-S_0))\mathbf{1}_{[S_1>t-S_0]}f(\beta(t-S_0-S_2))\mathbf{1}_{[S_3>t-S_0-S_2]}f(\beta(t-S_0-S_2))\mathbf{1}_{[S_4>t-S_0-S_2]}\mathbf{1}_{[S_2<t-S_0]}\mathbf{1}_{[S_0<t]} + \cdots, \end{aligned} \tag{9}$$

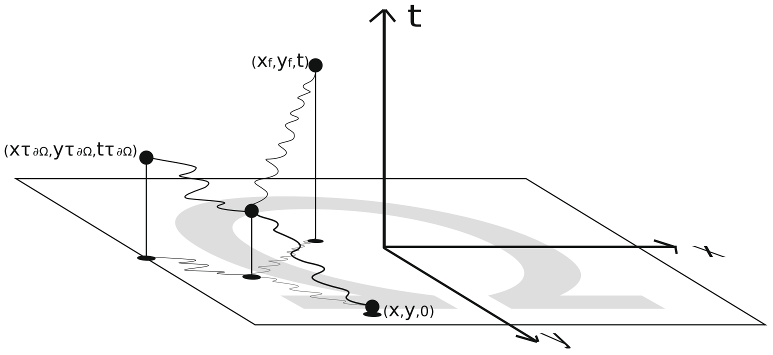

Fig. 1. A picture illustrating a typical branching process in 2D. The random tree starts at $(x, y, 0)$, and evolves in time inside the domain Ω. After some random exponential time, less than the final time, t, the path splits into two independent paths: one of them reaches the final time, t, while the other one touches upon the boundary, $\tau_{\partial\Omega}$ being the first exit time out of the domain Ω. Both of them do not suffer any further splitting.

where $S_0, S_1, S_2, \ldots, S_i, \ldots$ are independently generated random times, picked up from the exponential probability density $p(S) = c\exp(-cS)$. Note that in Eq. (9) each term contributes partially to the full solution.

A picture of such method, using branching stochastic processes, can be given as follows: a sufficiently high number of random exponentially distributed times, S_i, is first generated, so that their sum is less than or equal to the final time, t. For every random time, the given stochastic process solution of (3) is split into as many branches as those corresponding to the power of the nonlinearity. They start from the point where the previous stochastic process was at time S_i, and continue along independent paths until the next occurrence, S_{i+1}, takes place. Whenever one of these possible branches reaches the final time, t, the initial value, f, is evaluated at the position where the stochastic process was located. The solution is finally reconstructed multiplying all contributions coming from each branch. For the purpose of illustration, in Fig. 1 a 2D picture is shown, correspondingly to only two branches. Note that every branching process can be seen as a random tree with the space point $\mathbf{x}$ as its root. Therefore, in the probabilistic representation of solutions to nonlinear PDEs, a random tree plays a similar role as a single random path does in the linear case.

In case of a boundary value problem, with the boundary data $u(\mathbf{x}, t)|_{\mathbf{x}\in\partial\Omega} = g(\mathbf{x}, t)$, a similar representation holds, i.e.,

$$u(x,t) = E\left[\prod_{i=1}^{k(\omega)} \left\{ f(x_i(t,\omega))\mathbf{1}_{[t<\tau_{\partial\Omega_i}]} + g(\beta_i(\tau_{\partial\Omega_i}), t - \tau_{\partial\Omega_i})\mathbf{1}_{[t>\tau_{\partial\Omega_i}]} \right\}\right], \tag{10}$$

where $\tau_{\partial\Omega_i}$ is again the first exit time of the stochastic process $\boldsymbol{\beta}_i(\cdot)$.

3. The numerical method

The algorithm consists of three steps. To illustrate how it works, in Fig. 2 a sketch was plotted where such steps are shown for a two-dimensional problem. The first step is computing the solution at a few points by a probabilistic Monte Carlo-type method based on averaging over certain random trees, see Eq. (6). This is done on some chosen (either physical or artificial) "interfaces", located inside the space-time domain $D := \Omega \times [0, T]$, where $\Omega \subset \mathbf{R}^n$. In the following, such interfaces are obtained, for simplicity, partitioning the domain into subdomains like $D_i := [x_{i-1}, x_i] \times \Omega_0 \times [0, T]$, being $\Omega_0 \subset \mathbf{R}^{n-1}$. For instance, in $\mathbf{R}^2$ this corresponds to divide the domain in slices where the interfaces are parallel to y-axis. Once the solution has been computed, the second step is interpolating on such points, considered as interpolation nodes, thus obtaining continuous approximations of interfacial values of the solution. The third step, finally, consists in computing the solution inside each subdomain, which task can be assigned to separate processors. This can be realized resorting to local solvers, which may use classical numerical methods, such as finite differences or finite elements methods.

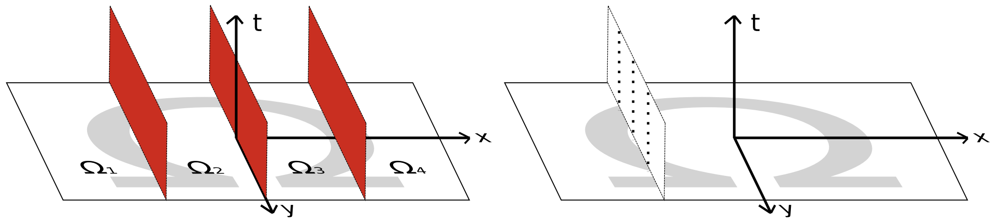

Fig. 2. A sketch illustrating the main steps of the algorithm in 2D: The figure on the left shows how the domain decomposition is done in practice. The figure on the right shows the points where the solution is computed probabilistically; these are used afterwards as nodal points for interpolation.

3.1. Probabilistic part

The purpose of this step is to compute the sought solution at a few single points, inside the space-time domain. Computing the solution at a high number of points so to cover a full computational domain is also possible but is exceedingly expensive, even though this approach could be pursued when the number of the available processors is extremely high. This can be done assigning the task of computing the solution at a set of points to different processors. The Monte Carlo method, in fact, is capable of fully exploit massively parallel architectures, and more, it is scalable to an arbitrary number of processors and is naturally fault tolerant.

When the parabolic equations are linear, a given number of random paths have to be generated, which obey the stochastic differential equation in (3), and track them until they either touch the boundary for the first time or reach a prescribed final time, t. The former case occurs in initial-boundary value problems (e.g. with Dirichlet and/or Neumann boundary conditions), while the latter case occurs in both a purely initial value problem, and a initial-boundary value problems. The solution to the equation at a given point, $(\mathbf{x}, t)$, can then be obtained by means of the Feymann-Kac formula in (4) or (2). In practice, the expected value is replaced by an arithmetic mean, since we must deal with a finite sample size, N. An alternative strategy to evaluate the solution was proposed in [30] for initial-boundary problems, which requires generating a random exponential time, S, obeying the probability density $P(S) = c\exp(-cS)$ for every random path. Then, depending on whether $S < t$ or not, the given path $\beta(t)$, contributes or not to the solution. Therefore, solution is computed as

$$u(\mathbf{x}, t) = E[f(\boldsymbol{\beta}(t))]. \tag{11}$$

In practice, the expected value above must be replaced necessarily by a finite sum, and moreover the stochastic paths are actually simulated resorting to suitable numerical schemes. Thus, approximately,

$$u(\mathbf{x}, t) = \frac{1}{N}\sum_{j=1}^{N} f\left(\beta_j^*(t)\right), \tag{12}$$

where N is the sample size, and β^* is the stochastic path with discretized time. Such a discretization procedure unavoidably introduces two sources of numerical error. The first one is the pure Monte Carlo statistical error, which it is known to be of order $O\left(1/\sqrt{N}\right)$ when N goes to infinity. The second error is due to the fact that the ideal stochastic path, $\beta_j(\cdot)$, has to be approximated, discretizing time, by some numerical scheme yielding the paths $\beta_j^*(\cdot)$. The truncation error made in solving numerically the stochastic differential equation (3), obviously depends on the specific scheme chosen, see [24], e.g. Among these are the Euler scheme, which was used here to simulate numerically Eq. (3). Such scheme is well known to have a truncation error of order $O(\Delta t^{\alpha})$, where $\alpha = 1/2$ or $\alpha = 1$ depending on whether the scheme being of the "strong" or "weak" type, respectively [24].

For the case of a boundary value problems, a new source of numerical error should be taken into account. In fact, consider, for the purpose of illustration, the Dirichlet problem for the one-dimensional heat equation, in presence of a constant sink term, $c > 0$,

$$\begin{aligned} &\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} - cu, \quad a < x < b, \quad t > 0, \\ &u(a, t) = 0, \quad u(b, t) = 0, \\ &u(x, 0) = f(x). \end{aligned} \tag{13}$$

The solution can be computed as

$$u(\mathbf{x}, t) = \frac{1}{N}\sum_{j=1}^{N} f(\beta_j^*(t))\mathbf{1}_{[S_j > \tau_\Omega]}. \tag{14}$$

In Fig. 3, we sketched the three possible scenarios the random paths $\beta_j^*(t)$ can undergo. Note that for the random paths of the type labelled with (3) in Fig. 3, it is required to evaluate precisely the first exit time out of the boundary. Such a task is however by far nontrivial, since $\tau_{\partial\Omega}$, in general, will be estimated numerically, and hence will be affected by numerical errors. Indeed, numerical experiments show that the error in estimating it may dominate over the other sources of numerical errors, and is therefore of paramount importance to assess accurately such a quantity.

In practice, the probability that a given approximate path exits the boundary between two consecutive time steps, is nonzero, and then it is possible that the true exit time might be overlooked. This circumstance has been pointed out in several occasions, see, e.g. [32,11,14].

In [20], it was estimated that the error made in evaluating (11) is of order $O(\Delta t^{1/2})$, being Δt the time step used in solving numerically the stochastic differential equation (3) by means of the Euler scheme. To reduce such an error it becomes crucial to evaluate accurately the first exit time, adopting suitable numerical strategies. Among the various possibilities considered in the literature, we chose to implement that proposed in [26] for one-dimensional problems, which is based on a theoretical approximation of the exit probability. To solve two-dimensional problems on the square, as we did in this paper, the value of the exit probability on Ω has been taken as the maximum among the four hitting probabilities that a trajectory first exits the

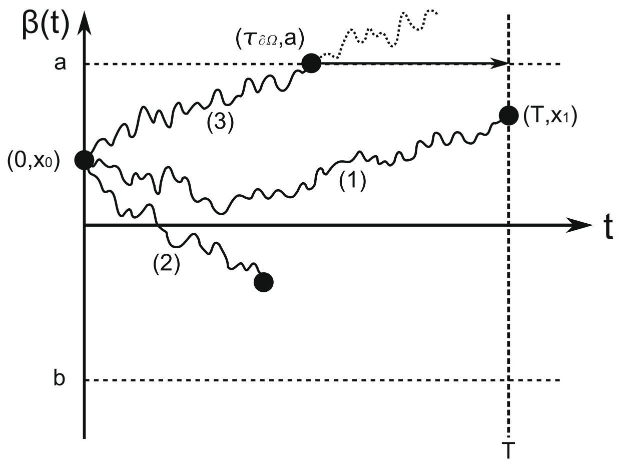

Fig. 3. The three possible patterns a random path may follow for the one-dimensional problem in (13). In (1), the generated exponential random time turns out to be greater than the final time, T; in (2), instead, the random time is smaller than T; in (3), the first exit time $\tau_{\partial\Omega}$ is smaller than both, the random and the final time.

four possible boundary-sides. This consists of an approximation of the true two-dimensional exit probability, but it suffices in order to achieve a numerical error now well below the statistical error.

3.1.1. Computational complexity for the nonlinear problem

In [5], the computational time required by the probabilistic part of the algorithm, when solving an initial value problem for a nonlinear one-dimensional PDE, has been estimated. More precisely, it was assumed that N random trees, initially located at $(x, 0)$, were generated and followed until they reached a prescribed final time, t. The same can be done for higher-dimensional PDEs when the solution at a single point, say $(\mathbf{x}, t)$, can be evaluated generating N random trees, starting at $(\mathbf{x}, 0)$, until they reach a prescribed final time. As in the one-dimensional case, the random tree associated to the nonlinear term u^m requires creating m branches every time a "splitting event" occurs, and this according to an exponential distribution. The latter is due to and governed by the linear term, $-cu$.

The computational time can be measured in terms of the number of iterations in time required to fully generate a random tree with k branches surviving at the final time, t. Note that for a boundary value problem, the computational time is smaller, because there are some trees whose branches may hit the boundary, and hence are stopped before reaching the prescribed final time. Then, the estimated computational time for a purely initial value problem can be used as an upper bound for the corresponding boundary value problem.

If t_c is the time spent per iteration, such computational time can be estimated as $kt_c t/\overline{\Delta t_s}$. Here $\overline{\Delta t_s}$ is the average value of the time step, and was shown in [5] (in the one-dimensional case) to be

$$\overline{\Delta t_s} = \frac{1 - e^{-c\Delta t}}{c}, \tag{15}$$

where Δt denotes the time step chosen to solve numerically the associated stochastic differential equation in (3). In case of N random trees, the average computational time, t_b (b standing for "branching"), turns out to be

$$t_b = N \sum_{k=1}^{\infty} k t_c \frac{t}{\overline{\Delta t_s}} P(k), \tag{16}$$

where $P(k)$ is the probability of finding a random tree with k branches. Such a probability was evaluated in [5] by first enumerating and then summing up the various probabilities of having k branches in the final configuration, and was given by

$$P(k) = (k - m + 1)!^{(m-1)} \frac{1}{N_e!(m-1)^{N_e}} e^{-cT} [1 - e^{-c(m-1)T}]^{N_e}, \tag{17}$$

where $N_e = (k-1)/(m-1)$, and the symbol $r!^{(s)}$ denotes the "multifactorial", or sth factorial, defined recursively as $r!^{(s)} = 1$ if $0 \leqslant r < s, r(r-s)!^{(s)}$ if $n \geqslant s$.

When $m = 2$ or 3, estimates t_b of practical use, asymptotically valid as $T \to +\infty$, were obtained,

$$t_b \leqslant N t_c \frac{T}{\overline{\Delta t_s}} e^{cT}, \quad \text{when } m = 2,$$

$$t_b \leqslant N t_c \frac{T}{\overline{\Delta t_s}} e^{2cT}, \quad \text{when } m = 3. \tag{18}$$

In the general case of $m \in \mathbf{N}, m > 3$, it was obtained, approximately,

$$t_b = O\left(Nt_c \frac{t}{\Delta t_s} \exp\left\{\frac{3m-5}{2}ct\right\}\right). \tag{19}$$

All this was obtained in [5] for the one-dimensional case, and can be trivially generalized to the n-dimensional case. In fact, the computational time depends on the total number of stochastic differential equations in (3), which increases linearly with the number of space dimensions.

As it was mentioned before for the linear case, in order to evaluate (6) we resort to numerical simulations of the Monte Carlo type, considering a finite size sample, N. In practice, we replace the expected value with an arithmetic mean, which is known to provide the best unbiased estimator to it [22]. The error made in doing so is statistical in nature and of the order of $N^{-1/2}$. However, it turns out that when we evaluate numerically functionals of a branching process, the fluctuations around the mean are often non-Gaussian (see [18,35], e.g.). For this reason, a large deviation analysis was conducted to assess the reliability of our numerical results. This has been done following the numerical strategy put forth in [31] to analyze simulated data. The deviation function estimated from the data obtained solving Example B of Section 5 at the single point $\mathbf{x} = (0, 0.5)$ and $t = 0.5$, is depicted in Fig. 4. In the inset the global behavior of the "deviation function" is shown for a full range of values, while in the larger figure the plot is expanded around the mean value. Recall that, from the theory of large deviations the probability distribution is known to decay exponentially fast as $P \asymp \exp(-NI(x))$, for values both larger or smaller than the empirical mean, $I(x)$ being the deviation function and N the sample size. Here $\asymp$ means logarithmic equivalence [31]. The sample size required to attain a given statistical error can be estimated computing the inverse of

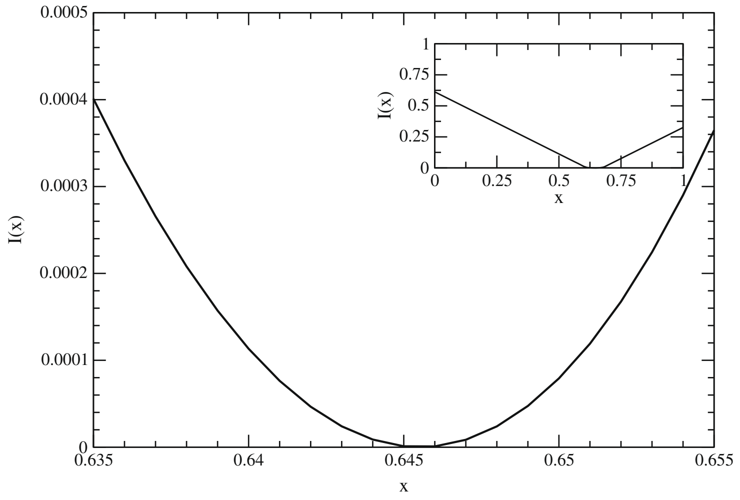

Fig. 4. The behavior of the deviation function for a sample size $N = 10^6$, obtained analyzing the simulated data when solving numerically Example B at the single point $\mathbf{x} = (0, 0.5)$, and $t = 0.5$.

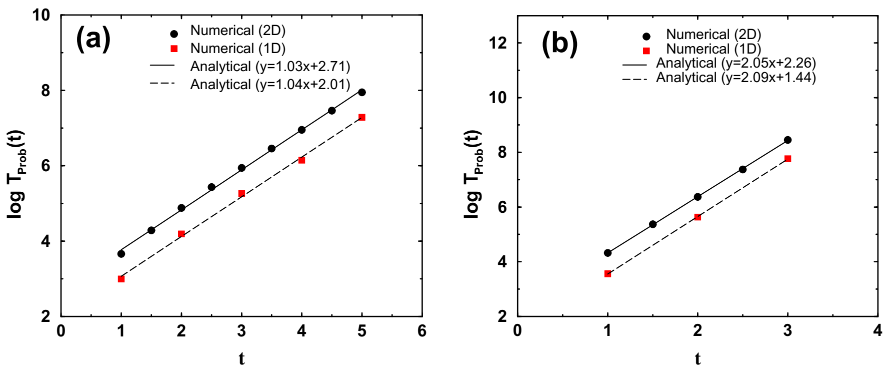

Fig. 5. Comparison between the computational time spent in solving an initial value problem for the nonlinear PDE in (8) at a single point ($\mathbf{x} = 0$), and the estimated computational time obtained theoretically from Eq. (18). This has been done in (a), for a purely quadratic nonlinearity, and in (b), for a purely cubic nonlinearity. In both cases the comparison has been done for one and two-dimensional problems. Other parameters used are $N = 10^6$, and $c = 1, \Delta t = 10^{-2}$.

the deviation function evaluated at the point where the deviation function departs from its minimum value, that is the expected mean value. For the present case, to obtain the mean value with a statistical error of order 10^{-3} with high probability, N should be higher than 10^4. In fact, with $N = 10^5$, the probability of obtaining a value larger than the empirical mean plus the statistical error is already $P \approx 10^{-3}$. Therefore, choosing a sample size of order 10^6 in our numerical examples suffices to guarantee a reliable approximation. Moreover, in some cases we were able to check the actual numerical error by a direct comparison with the analytical solution, confirming that this size indeed suffices in all our test problems.

In Fig. 5, a comparison between the analytical results obtained in (18) and the measured computational time is shown as function of the final time, t in logarithmic scale (for the y-axis). Here the computational time, spent in solving probabilistically the corresponding initial value problem for the nonlinear PDE in (8), at a single point $\mathbf{x} = 0$, has been measured. Fig. 5(a) shows the results corresponding to the case with only α_2 different from zero in Eq. (8), and Fig. 5(b) those corresponding to α_3, that is a purely quadratic and cubic nonlinearity, respectively. The initial condition was chosen to be $f(\mathbf{x}) = \exp(-\mathbf{x}^2/4\sigma)/\sqrt{4\pi\sigma}$, with $\sigma = 1$. In all simulations below, the value of σ was chosen in such a way that $\max_{\mathbf{x}\in\mathbf{R}^n} f(\mathbf{x}) < 1$. This is to guarantee convergence of the expansion (9). Note that the computational time spent by the probabilistic part of the algorithm for the two-dimensional problem is almost twice the time spent for solving the one-dimensional problem. This is in agreement with the theoretical considerations above, and confirms that, in general, increasing the dimensionality of the problem merely affects the computational time, in that the number of stochastic differential equation (3) to be solved increase correspondingly.

3.2. Interpolation in space-time

Let assume that we have already computed the values of the sought solution at some points on the interfaces $x = x_i$, by the previous Monte Carlo approach. These are the points $(x_i, \mathbf{y}_j, t_k)$, where $\mathbf{y}_j \in \Omega_0$, for every fixed i, and very few j's and k's. A number of numerical schemes can be adopted to interpolate in the $n-1$ dimensional space Ω_0. The simplest method of obtaining multivariate interpolation is to consider a univariate method and derive from it a multivariate method by tensor product. In practice, given $n-1$ set of points, the tensor product interpolation finds the corresponding interpolation coefficients solving repeatedly univariate interpolation problems as described in [16]. For the two-dimensional examples in Section 5, a tensor product interpolation based on cubic splines was adopted [6]. Here the nodal points are uniformly distributed on Ω_0, and a not-a-knot condition has been imposed, which means imposing continuity of the third derivative at the boundary. When the number of nodal points, n, is the same along each dimension, interpolating at a single point (y_j, t_k) requires $n+1$ spline calculations to obtain the spline coefficients, and then evaluating the spline value at $n+1$ points. The computational cost for calculating the spline coefficients is known to be of order $O(n)$, while for evaluating the spline value it is $O(\log n)$. The interpolation error when the interpolating function is sufficiently smooth (C^8 at least) is of order $O(h^4 + l^4)$ [33], where h and l are the widths of the interpolating grid in the y and t axes, respectively.

In Fig. 6, the pointwise numerical error made when interpolating at the interface $(0, y, t)$ has been shown. The nodal points were obtained solving probabilistically the two-dimensional KPP equation on the square domain $\Omega = [-1,1]\times[-1,1]$, with the initial condition $u(x, y, 0) = \left[1 + (\exp(x-y))/2\sqrt{3}\right]^{-2}$, and boundary conditions

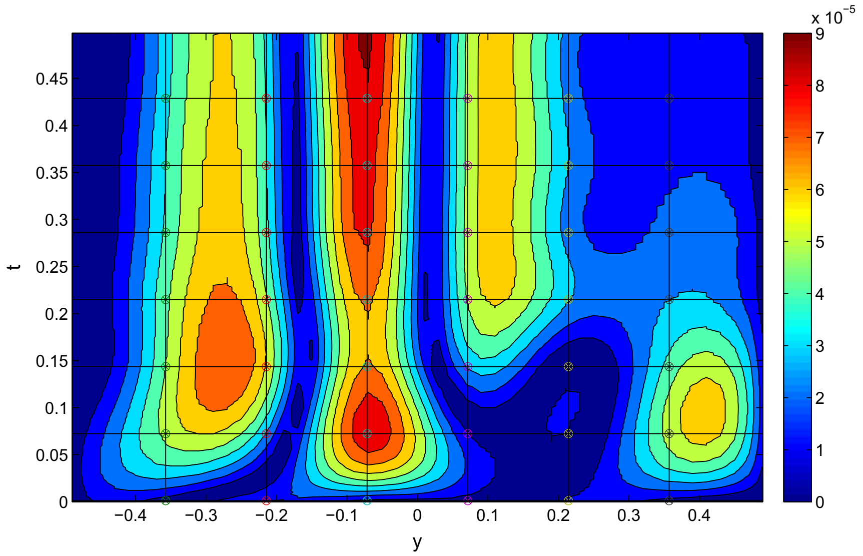

Fig. 6. Contour plot showing the pointwise numerical error obtained when interpolating at the interface $(0, y, t)$ using 8 nodal points along y, and t. Such points are obtained solving probabilistically a Dirichlet boundary value problem for the KPP equation in 2D. The parameters used are as in Fig. 5.

$$u(x,y,t)|_{\partial\Omega} = \left(1 + e^{\frac{\frac{x-y}{\sqrt{2}}-\frac{5}{\sqrt{6}}t}{\sqrt{6}}}\right)^{-2}\Bigg|_{\partial\Omega}. \tag{20}$$

The error can be evaluated precisely since for this problem an analytical solution is known, i.e.,

$$u(x,y,t) = \left(1 + e^{\frac{\frac{x-y}{\sqrt{2}}-\frac{5}{\sqrt{6}}t}{\sqrt{6}}}\right)^{-2}. \tag{21}$$

Note that the numerical error due to interpolation lie already below that affecting the nodal points (marked on the Figure), which have been computed probabilistically, and subject to the numerical error described in the previous section.

3.3. Local solver

Once that continuous interfacial approximations of the solution have been obtained upon interpolation on the previously computed nodes (by Monte Carlo), we can solve the original problem *on each* subdomain, D_i, *independently* of each other, since a full decoupling has been realized. Hence, the numerical treatment on each subdomain can be accomplished by a local solver, which can also be different from all the others. In the numerical examples below, we used a solver based on the LU factorization.

4. Reducing the computational complexity of the probabilistic part by pruning the trees

The high computational cost of the probabilistic part of the algorithm is clearly displayed by the estimates in (18), characterized by terms growing exponentially in time. Indeed, this is a serious problem especially when the final time, t, is large, and no boundary conditions are prescribed. Clearly, the reason is that generating branching processes with a large number of branches implies a strong computational effort. However, in the following we show that such an effort can be contained, mainly because the contribution of the branching processes to the global solution becomes negligible quite soon, i.e., in general, after only a low number of splitting events. For such a reason, a suitable "pruning" of the trees can be accomplished, increasing dramatically in such a way the performance of the overall algorithm. In the following we show in practice how this works.

Eq. (9) shows that the solution $u(\mathbf{x},t)$ can be obtained by summing up the partial contributions in the configuration diagrams, consisting of random trees with an arbitrary number of branches, say k. Therefore, Eq. (6) can be rewritten as

$$u(\mathbf{x},t) = \sum_{k=1}^{\infty}\frac{1}{c^{k-1}}u^{(k)}(\mathbf{x},t), \tag{22}$$

where $u^{(k)}(\mathbf{x},t) = E\left[\prod_{i=1}^{k} f(\mathbf{x}_i)\right]$, see Eq. (9). It can be proved that

$$u^{(k)}(\mathbf{x},t) \leqslant P(k)\left[\max_{\mathbf{x}\in\mathbf{R}^n} f(\mathbf{x})\right]^k. \tag{23}$$

The proof for arbitrary values of m in Eq. (8) is rather cumbersome. Here, we consider only the case with a purely quadratic nonlinearity, for the purpose of illustration. From Eq. (9) follows

$$u^{(2)}(\mathbf{x},t) = \int_0^t dS_0 ce^{-cS_0}\int_{-\infty}^{+\infty} d\mathbf{x}_0 p(\mathbf{x},S_0,\mathbf{x}_0,0)\times\left(e^{-c(t-S_0)}\int_{-\infty}^{+\infty} d\mathbf{y}_1 f(\mathbf{y}_1)p(\mathbf{x}_0,t-S_0,\mathbf{y}_1,0)\right)^2. \tag{24}$$

Here p is the Green's function, satisfying

$$\frac{\partial p}{\partial t} = Lp, \quad x\in\Omega, \quad t>\tau, \tag{25}$$

$$p(\mathbf{x},0,\mathbf{y},\tau) = \delta(\mathbf{x}-\mathbf{y})\delta(\tau), \tag{26}$$

where L is the linear elliptic operator in (1). Changing S_0 into $S_0' := t - S_0$, we obtain

$$u^{(2)}(\mathbf{x},t) = e^{-ct}\int_0^t dS_0' ce^{-cS_0'}\int_{-\infty}^{+\infty} d\mathbf{x}_0 p(\mathbf{x},t-S\prime_0,\mathbf{x}_0,0)\xi^2(\mathbf{x}_0,S_0'), \tag{27}$$

where $\xi(\mathbf{x}_0,S_0')$ is given by

$$\xi(\mathbf{x}_0,S_0') = \int_{-\infty}^{+\infty} d\mathbf{y}_1 f(\mathbf{y}_1)p(\mathbf{x}_0,S\prime_0,\mathbf{y}_1,0). \tag{28}$$

Note that ξ satisfies the Cauchy problem for the linear parabolic partial differential equation (1) with $c=0$ and initial condition $\xi(\mathbf{x}_0,0) = f(\mathbf{x}_0)$. Therefore $\xi(\mathbf{x}_0,S_0') \leqslant \max_{\mathbf{x}_0\in\mathbf{R}^n} f(\mathbf{x}_0)$, and hence

$$u^{(2)}(\mathbf{x},t) \leqslant e^{-ct}(1-e^{-ct})\eta^2, \tag{29}$$

where $\eta = \max_{\mathbf{x}\in\mathbf{R}^n} f(\mathbf{x})$. Recall that, from Eq. (17), the probability of branching with $k=2$ is precisely $P(2) = e^{-ct}(1-e^{-ct})$, and hence

$$u^{(2)}(\mathbf{x},t) \leqslant P(2)\eta^2. \tag{30}$$

Let now describe the steps needed to evaluate the contribution to the global solution of random trees with $k=3$ branches. Obtaining a configuration diagram with three branches requires generating five exponential random times, S_i, with $i=1,\dots,4$, such that

$$\begin{aligned} &S_0 < t, \quad S_1 > t-S_0, \quad t-S_0 > S_2 > S_0, \\ &S_3 > t-S_0-S_2, \quad S_4 > t-S_0-S_2. \end{aligned} \tag{31}$$

It was shown in [5] that there are two configuration diagrams satisfying the conditions above, and both of them contribute equally to the global solution. In the following, we focus on one of them, the overall partial contribution being then multiplied by two. From Eq. (9), we have

$$\begin{aligned} u^{(3)}(\mathbf{x},t) = 2\int_0^t dS_0 ce^{-cS_0}\int_{-\infty}^{+\infty} d\mathbf{x}_0 p(\mathbf{x},S_0,\mathbf{x}_0,0)\ e^{-c(t-S_0)}\int_{-\infty}^{+\infty} d\mathbf{y}_1 f(\mathbf{y}_1)p(\mathbf{x}_0,t-S_0,\mathbf{y}_1,0) \\ \times\int_0^{t-S_0} dS_2 ce^{-cS_2}\int_{-\infty}^{+\infty} d\mathbf{x}_1 p(\mathbf{x}_1,S_2,\mathbf{x}_0,0)\left(e^{-c(t-S_0-S_2)}\int_{-\infty}^{+\infty} d\mathbf{y}_2 f(\mathbf{y}_2)p(\mathbf{x}_1,t-S_0-S_2,\mathbf{y}_2)\right)^2, \end{aligned} \tag{32}$$

Changing the variables S_0 and S_2 into $S'_0 = t-S_0$ and $S'_2 = t-S_0-S_2$, respectively, we get

$$u^{(3)}(\mathbf{x},t) = 2e^{-ct}\int_0^t dS'_0 ce^{-cS'_0}\int_{-\infty}^{+\infty} d\mathbf{x}_0 p(\mathbf{x},t-S'_0,\mathbf{x}_0,0)\xi(\mathbf{x}_0,S'_0)\ \int_0^{S'_0} dS'_2 ce^{-cS'_2}\int_{-\infty}^{+\infty} d\mathbf{x}_1 p(\mathbf{x}_0,S'_0-S'_2,\mathbf{x}_1,0)\xi^2(\mathbf{x}_1,S'_2). \tag{33}$$

Using Eq. (27), such equation can be rewritten in terms of $u^{(2)}$ as

$$u^{(3)}(\mathbf{x},t) = 2e^{-ct}\int_0^t dS'_0 c\int_{-\infty}^{+\infty} d\mathbf{x}_0 p(\mathbf{x},t-S'_0,\mathbf{x}_0,0)\xi(\mathbf{x}_0,S'_0)u^{(2)}(\mathbf{x}_0,S'_0). \tag{34}$$

As in the case of two branches, we have $\xi(\mathbf{x}_1,S'_2) \leqslant \max_{\mathbf{x}_1\in\mathbf{R}^n} f(\mathbf{x}_1)$. It then follows from Eq. (30) that

$$u^{(3)}(\mathbf{x},t) \leqslant 2\eta^3 e^{-ct}\int_0^t dS'_0 ce^{-cS'_0}(1-e^{-cS'_0}) = e^{-ct}(1-e^{-ct})^2\eta^3. \tag{35}$$

Since the probability of finding three branches is $P(3) = e^{-ct}(1-e^{-ct})^2$, we can conclude that $u^{(3)} \leqslant P(3)\eta^3$. By simply repeating this procedure, used to obtain $u^{(3)}$ in terms of $u^{(2)}$, a general expression can be readily derived for the partial contribution of $k+1$ branches, $u^{(k+1)}$, as a function of $u^{(k)}$, and the result is

$$u^{(k+1)} = ke^{-ct}\int_0^t dS'_0\int_{-\infty}^{+\infty} d\mathbf{x}_0 p(\mathbf{x},t-S'_0,\mathbf{x}_0,0)\xi(\mathbf{x}_0,S'_0)u^{(k)}(\mathbf{x}_0,S'_0). \tag{36}$$

The general estimate in (23) now follows induction. In fact, assuming that $u^{(k)} \leqslant P(k)\eta^k$, we derive from Eq. (36)

$$u^{(k+1)} \leqslant k\eta^{k+1}e^{-ct}\int_0^t dS'_0 ce^{-cS'_0}\left(1-e^{-cS'_0}\right)^{k-1} = \eta^{k+1}e^{-ct}(1-e^{-ct})^k. \tag{37}$$

Therefore, $u^{(k+1)} \leqslant \eta^{k+1}P(k+1)$, as it had to be proved.

In Fig. 7, the partial contributions, $u^{(k)}(\mathbf{0},t)$, obtained solving numerically the KPP equation in 1D and 2D at a single point ($\mathbf{x} = \mathbf{0}$), by means of random trees, are shown. This has been done for two different values of the final time, t. Moreover, the results of the numerical simulations are compared with the theoretical estimates of the partial contributions as given in (23). Note that the results of the numerical simulations lie always below the theoretical values, which fact proves that the estimates actually are overestimates as expected.

It is remarkable that the partial contributions to the global solution decay very rapidly as the number of branches increases. Therefore, we expect that truncating the expansion in (22) to only a certain number of branches, might not affect appreciably the result. Such truncation can be seen as "a pruning" of the full random tree, which amounts to keep only few trees possessing a certain number of branches. Such a procedure has been used in literature before, but mainly to expand the set of initial-conditions for which a probabilistic representation can be found (see [13], e.g.). Performing such a pruning, the additional truncation error

$$\varepsilon := \sum_{k=k_{max}+1}^{\infty} u^{(k)} \tag{38}$$

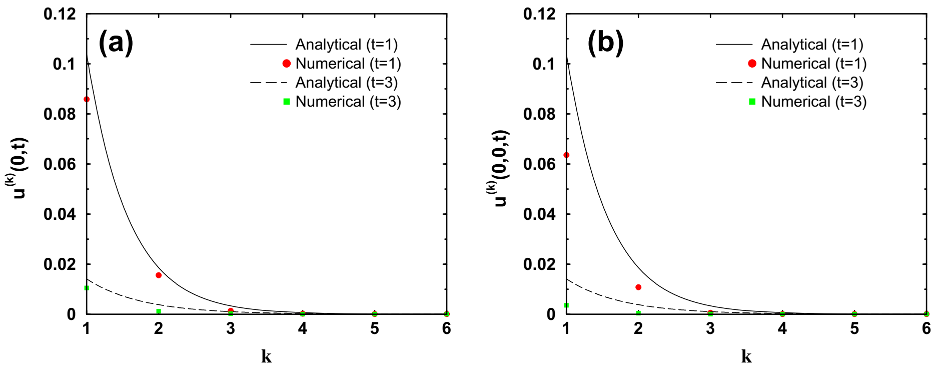

Fig. 7. Partial contributions to the global solution of the KPP equation (a) in 1D, and (b) in 2D, as a function of the number of branches. This has been done for the initial condition $f(\mathbf{x}) = \exp(-\mathbf{x}^2/4\sigma)/\sqrt{4\pi\sigma}$, for two different values of the final time, $t = 1$ and $t = 3$. The other parameters are $N = 10^6, c = 1$, and $\sigma = 1$.

appears, k_{max} denoting the maximum number of branches taken into account in expansion (22). Such an error can be bounded from above. This follows from Eq. (23) and we obtain

$$\varepsilon \leqslant \sum_{k=k_{max}+1}^{\infty} P(k)\eta^k \tag{39}$$

and being $\eta < 1$, then it holds that

$$\varepsilon \leqslant \eta^{k_{max}+1} \sum_{k=k_{max}+1}^{\infty} P(k). \tag{40}$$

When the nonlinearity is purely quadratic, replacing $P(k)$ with the corresponding analytic form of the probability in Eq. (17), the series above can be worked out, yielding

$$\varepsilon \leqslant \eta^{k_{max}+1}(1 - e^{-ct})^{k_{max}}. \tag{41}$$

In Fig. 8, the numerical error obtained solving an initial value problem for the KPP equation in 1D is shown as a function of k_{max}. We chose the initial condition $f(x) = \exp(-x^2/4\sigma)/\sqrt{(4\pi\sigma)}$ with $\sigma = 0.1$. Since an analytical solution for the corresponding problem is not known, we compared it with the numerical solution obtained by a finite difference implicit method with a very fine space-time grid.

Let consider a time step Δt sufficiently small to guarantee that the error due to the discretization of Eq. (3) is smaller than the statistical error, ε_{stat}. It turns out that ε_{stat} is the dominant error in the numerical probabilistic evaluation of $u(\mathbf{x}, t)$ by

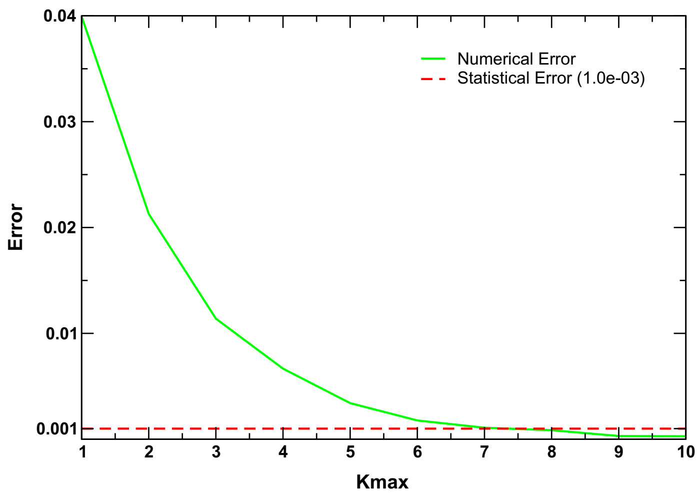

Fig. 8. Truncation error, ε, obtained solving an initial value problem for the KPP equation in 1D at $x = 0$ and $t = 2$, as a function of the maximum number of branches, k_{max}, taken into account. The initial condition was $f(x) = \exp(-x^2/4\sigma)/\sqrt{4\pi\sigma}$ with $\sigma = 0.1$. The other parameters are as in Fig. 7.

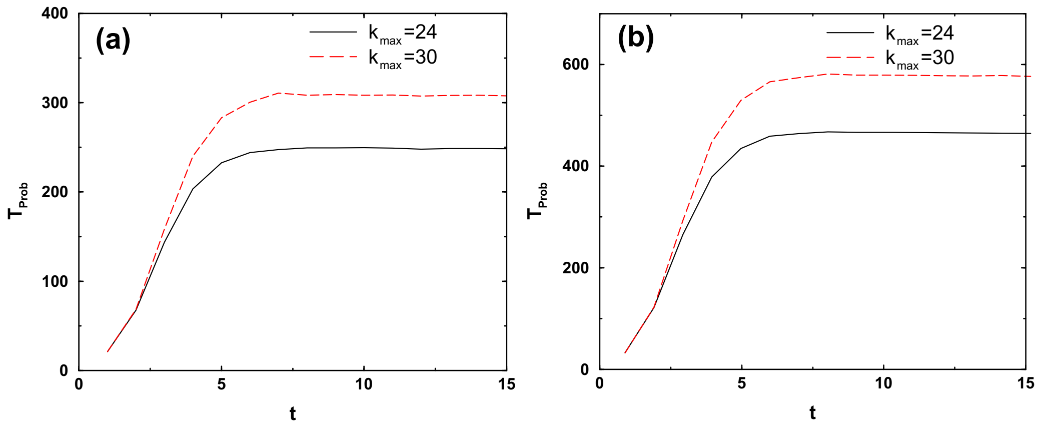

Fig. 9. Computational time as a function of the final time spent to solve numerically the KPP equation in: (a) 1D, and (b) 2D at a single point, **x** = 0. Here pruned random trees, with $k_{max} = 24$ and $k_{max} = 30$, have been generated. The parameters used are as in Fig. 7.

means of (22). This suggests that for any given sample size, should be possible to choose a (finite) value of k_{max} such that the truncation error ε lie well below ε_{stat}, and hence undetectable, in practice, in the numerical results. In other words, this means that the contribution to the solution of random trees with a number of branches k larger than k_{max} is negligible.

For the case of the KPP equation in n dimensions, with the initial condition $f(\mathbf{x}) = \exp(-\mathbf{x}^2/4\sigma)/\sqrt{(4\pi\sigma)}$, the right-hand side of Eq. (41) can be worked out analytically, and k_{max} can be obtained as a function of ε_{stat}. We have

$$k_{max} = \frac{\log \varepsilon_{stat} + \frac{1}{2}\log(4\pi\sigma)}{\log(1-e^{-ct}) - \frac{1}{2}\log(4\pi\sigma)}. \tag{42}$$

For the parameters chosen to provide the results plotted in Fig. 8, k_{max} can be computed, and it turns out to be approximately equal to 6. Note that the truncation error ε plotted in Fig. 8 becomes smaller than the statistical error (which is of order of 10^{-3} for the parameter values chosen) as k_{max} becomes larger than 6.

Clearly, pruning the trees is important since it allows to reduce appreciably the CPU time required to compute the solution probabilistically. In fact, as is shown in Fig. 9, in doing so the computational time becomes nearly constant after long times. The computational time required to solve probabilistically the KPP equation in 1D and 2D at a single point was measured as a function of the final time, pruning the trees with $k_{max} = 24$ and $k_{max} = 30$, respectively.

The explanation of such a behavior is that random trees whose total number of branches exceeds k_{max} in the simulations can be suddenly cut without affecting the results. In practice, they contribute to the computational time as they were trees having only k_{max} branches. Therefore, the computational time spent for long final times can be estimated as

$$t_b \approx k_{max} t_c \sum_{k=k_{max}+1}^{\infty} P(k) = k_{max} t_c (1-e^{-ct})^{k_{max}}. \tag{43}$$

Note that when $t \to +\infty, t_b \to k_{max} t_c$. This agrees with what can be observed in Fig. 9. In fact, the ratio between the asymptotic values of the computational times for the two values of k_{max} chosen above coincides approximately with the theoretically expected value, 30/24 (=1.25). Note that the results do not depend on the dimension, as was expected.

5. Numerical examples

In this section, we present some numerical examples for the two-dimensional case, on the square $\Omega = [-L, L] \times [-L, L]$, to illustrate the probabilistically induced domain decomposition (PDD) algorithm described in the previous sections. All simulations were carried out on the MareNostrum Supercomputer of the Barcelona Supercomputing Center, using up to 1,024 processors.

5.1. Computer science-related issues

In order to assess the performance of our method, a comparison was made solving the same problem by some other (classical) numerical schemes. The space-time domain as well as the subdomains in our decomposition being simple, we used the Crank–Nicolson (implicit) finite difference method. On the various uncoupled subdomains obtained by the PDD algorithm we used LAPACK, while the full domain solution was computed by ScaLAPACK, which is considered extremely efficient for the parallel solution of banded linear systems. In fact, the resulting matrix A associated to the linear algebraic problem to be solved is banded, with a bandwidth BW, say.

In the following, we keep fixed the space discretization of the two-dimensional finite difference computational mesh, choosing an equal number of computational nodes in both dimensions, that is $N_x = N_y = N$. Note that the bandwidth of the matrix A is N, being $A \in \mathbb{R}^{N^2 \times N^2}$. The memory consumption and the computational cost of both methods, the PDD and ScaLAPACK, can be estimated as a function of the number of processors, p. ScaLAPACK requires storing the full matrix in a block-column distribution format among the p processors [12]. When the matrix is not diagonally dominant, the LU factorization may be numerically unstable, and a partial pivoting with elimination might be necessary. In practice, this may require an additional storage to prevent a fill-in. Furthermore, reordering is often used to increase parallelism in the factorization. In [7,8], the local memory needed per processor, M_p, has been estimated, and shown to decrease proportionally to p, more precisely as

$$M_{p_{ScaLAPACK}} = \frac{N^2}{p}(4BW + 1). \tag{44}$$

The bandwidth is not affected by the number of processors involved, and for such a computational mesh it turns out to be always $BW = N$. Then, it follows

$$M_{p_{ScaLAPACK}} = 4\frac{N^3}{p} + \frac{N^2}{p}. \tag{45}$$

The computational cost of ScaLAPACK for solving the full banded linear system, $C_{ScaLAPACK}$, was also estimated theoretically in [7,8]. For our purpose, the upper-half bandwidth, k_u, and lower-half bandwidth, k_l, are equal, $k_l = k_u = BW$. It follows that

$$\begin{aligned} C_{ScaLAPACK} &\approx (4(2N)^2 + 6(2N))\frac{N}{p} + \left(\frac{23}{3}(2N)^3 + 12(2N)^2\right)\lfloor \log_2 p \rfloor + t_{comm} + 4\frac{N^3}{p} + 2\frac{N^2}{p} \\ &= \frac{16N^3 + 12N^2}{p} + \left(\frac{184}{3}N^3 + 48N^2\right)\lfloor \log_2 p \rfloor + t_{comm} + 4\frac{N^3}{p} + 2\frac{N^2}{p}, \end{aligned}$$

see [7,8]. Here t_{comm} represents the interprocessor communication cost, $t_{comm} \approx \mathcal{O}(N^2\lfloor \log_2 p \rfloor)$, which is due to the cyclic reduction that permits pivoting [21,9]. The last two terms are due to the computational time spent evaluating the coefficients of the matrix and the right-hand side, respectively. Note that the appearance of an intercommunication overhead may degrade the performance of the algorithm when the number of processors involved increases.

As for the PDD method, once the solution has been computed on the interfaces, the full domain can be split into p subdomains, and assigned to different processors. Therefore, each processor can be devoted only to the solution of its local linear system, whose banded associated matrix is smaller. In fact, the memory consumption per processor, including an extra fill-in space, will be considerably reduced. With LAPACK as local solver, the resources have been estimated theoretically [7,8], and amount to

$$M_{p_{PDD}} = \frac{N^2}{p} 3BW. \tag{46}$$

Since the subdomains are now fully independent of each other, the bandwidth corresponds to the local matrix on each subdomain (assumed to be all identical, for simplicity), hence N/p. Therefore,

$$M_{p_{PDD}} = 3\frac{N^3}{p^2}. \tag{47}$$

Note that, with a large number of processors, this is significantly smaller that obtained for ScaLAPACK, and the relative advantage increases linearly with p, being

$$\frac{M_{p_{PDD}}}{M_{p_{ScaLAPACK}}} = \frac{3N}{4N+1}\frac{1}{p}.$$

The local solver for the PDD method, being based on LAPACK for solving banded linear system, consists of the LU factorization followed by a forward/backward substitution. The computational cost is known to be of order of N^2BW^2 for the LU factorization, and N^2BW for the forward/backward substitution, N^2 being the size of the square matrix and $BW = N/p$. Hence, with the local solver LAPACK, the computational cost of the local solver of the PDD can be readily estimated,

$$C_{PDD} \approx \alpha_{opt}(4BW + 1)BW\frac{N^2}{p} + 3\frac{N^3}{p^2} + \frac{N^2}{p} = \frac{N^2}{p}\left(4\alpha_{opt}\frac{N^2}{p^2} + (\alpha_{opt} + 3)\frac{N}{p} + 1\right), \tag{48}$$

see [7,8]. The last two terms are again related to the computational time spent to compute the matrix coefficients and the right-hand side, respectively. Here the parameter $\alpha_{opt} < 1$ was introduced to account for the computational advantages gained when handling an optimized LAPACK library, after tuning conveniently the BLAS library for the specific hardware platform used.

In view of such results, it is worth pointing out some important differences in terms of memory consumption for both methods. In fact, while in ScaLAPACK the memory consumption decreases as p^{-1}, for the PDD it is proportional to p^{-2}. In practice, this allows the PDD algorithm to exploit at best the available computational resources, being capable to handle problems whose size can be much higher than ScaLAPACK can afford.

As for the computational cost, the PDD algorithm performs much better than ScaLAPACK. In fact, when $N \gg p$, the computational cost of the PDD method decreases as p^{-3} for large p, while that of ScaLAPACK decreases only as p^{-1}.

Due to the strong limitations of ScaLAPACK in terms of memory consumption, it is difficult to chose a convenient experimental range of values for an arbitrary number of processors, suitable to compare the scalability properties of both methods, when both are performing at best of theirs capabilities. Indeed, the optimal computational workloads for ScaLAPACK are in general rather modest compared to the PDD high performance, being in all cases severely underused.

To this purpose, we tried first to adjust carefully the parameter values to deal with the limitations imposed by ScaLAPACK. However, it happens that for this case the total computational workload is too low for the local solver of the PDD. The overall computational time is basically due to the probabilistic part of the algorithm (the Monte Carlo computations), which turns out to be independent of the number of processors, p. The reason should be found in the way the PDD algorithm is implemented in practice on a parallel computer. We assigned the task of computing each set of interfacial values to a different processor. When the number of processors is sufficiently large, assigning these independent tasks to different processors can be done in a one-to-one mapping, leaving only one processor idle in such a mapping (in fact, the number of interfaces turns out to be equal to the number of processors minus one). Moreover, the size of the local linear algebraic problem corresponding to each subdomain is reduced considerably, making the computational time spent by the local solver negligible compared to the time spent by the Monte Carlo and the interpolation parts of the algorithm. Therefore, the overall computational time of the algorithm tends to remain constant when the number of processors is sufficiently large. That time is however by far much smaller than the best time ever achieved by ScaLAPACK.

5.2. Numerical test problems

In this subsection we consider two numerical test problems.

Example A. In Table 1, we compared the results obtained by the PDD method and by ScaLAPACK, solving a Dirichlet boundary value problem for the KPP equation in 2D. The problem is given by

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - u + u^2 \quad \text{in } \Omega = [-L, L] \times [-L, L], \quad t > 0, \tag{49}$$

with boundary- and initial-conditions

$$u(x,y,t)|_{\partial\Omega} = g(x,y,t)_{\partial\Omega}\big|_{\partial\Omega}, \quad u(x,y,0) = \left(1 + e^{\frac{x-y}{2\sqrt{3}}}\right)^{-2}, \tag{50}$$

$g(x,y,t)$ being the known analytical solution to this problem,

$$g(x,y,t) = \left(1 + e^{\frac{\frac{x-y}{\sqrt{2}} - \frac{5}{\sqrt{6}}t}{\sqrt{6}}}\right)^{-2}. \tag{51}$$

The parameters chosen for the local solver were: $\Delta x = \Delta y = 2.5 \times 10^{-4}, L = 0.5$ for the space domain, and $\Delta t = 10^{-3}$ for the time domain. The solution $u(x,y,t)$ was computed at the final time $t = 0.5$. The time step used solving Eq. (3) was $\Delta t = 10^{-2}$.

The third column (ScaLAPACK) in Table 1 shows the overall computational time (in seconds) spent by ScaLAPACK using $p = 64, 128$, and 256 processors. The corresponding total time spent by the PDD algorithm is shown in the second column, being also displayed here the computational time required by the Monte Carlo part of the algorithm (T_{MC}) and the interpolation part (T_{INTERP}). Recall that the overall computational time is simply the sum of the time spent by the three parts of the algorithm, that is the Monte Carlo part, the interpolation part, and the local solver. The two methods were compared correspondingly to the same maximum error, 10^{-3}. With both algorithms the CPU time decreases as p increases, and this trend is more dramatic in the PDD algorithm. Note that for a large number of processors, the CPU time for the PDD algorithm tends to stabilize as was explained before. Clearly, the PDD outperforms ScaLAPACK, the differences being absolutely striking.

Table 1
Comparing the computational times spent by PDD and Scalapack in Example A.

Number of processors	PDD			ScaLAPACK
	T_{MC}	T_{INTERP}	T_{TOTAL}	T_{TOTAL}
64	2’ 17”	<1”	3’ 37”	–
128	2’ 18”	<1”	2’ 23”	7510’ 20”
256	2’ 18”	<1”	2’ 18”	5223’ 41”

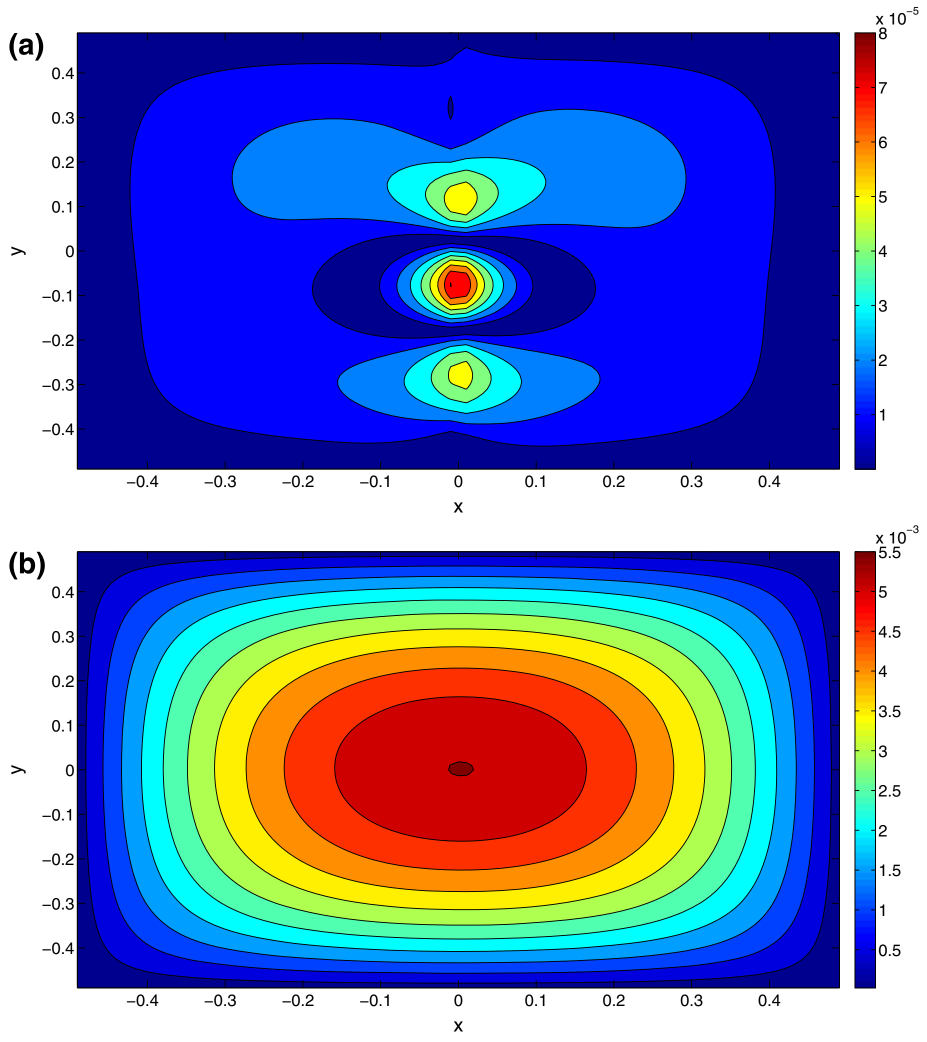

Fig. 10. Pointwise numerical error for the solution to the problem of Example A, at $t = 0.5$ in: (a) the PDD algorithm, and (b) ScaLAPACK.

In Fig. 10, contour plots for the pointwise numerical error made solving the problem of Example A, by both, PDD (a), and ScaLAPACK (b) are shown. Since the LU decomposition implemented in LAPACK is not suited to be parallelized, a reordering with additional permutations on the input matrix is done in ScaLAPACK, aiming to extract concurrency from the algorithm. Therefore, in practice, ScaLAPACK requires more internal operations to solve the full problem than LAPACK does. As a consequence, the expected final numerical error simply due to the propagation of roundoff errors in the algorithm may be higher than that obtained using LAPACK. This may explain what can be observed in Fig. 10.

The previous results were obtained aiming to compare the performance of the PDD algorithm with that of another (freely available) method. This was done adjusting the size of the domain making it suitable for the classical method. This approach prevented, however, to exploit at best the PDD algorithm. In the following, we show some results obtained only using the PDD algorithm, because ScaLAPACK was unable to conduct the same task with the available computational resources.

Table 2 refers to the problem of Example A and shows the computational time spent by the PDD algorithm when runned with 128,256, and 512 processors. This has been done keeping fixed the space domain, choosing now $L = 1$.

Table 2
Computational time spent by the PDD algorithm in Example A.

Number of processors	PDD		
	T_{MC}	T_{INTERP}	T_{TOTAL}
128	4’ 53”	<1”	214’ 17”
256	5’ 00”	<1”	54’ 11”
512	4’ 55”	<1”	15’ 58”

Table 3
Computational time spent by the PDD algorithm in Example B.

Number of processors	PDD		
	T_{MC}	T_{INTERP}	T_{TOTAL}
128	1' 05"	<1"	209' 44"
256	1' 05"	<1"	53' 10"
512	1' 06"	<1"	12' 13"
1024	1' 08"	<1"	3' 59"

Table 4
Computational time spent by the PDD algorithm in Example B for a rectangular domain.

N. CPUs	PDD		
	T_{BBM}	T_{INTERP}	T_{TOTAL}
128	1' 07"	<1"	903' 55"
256	1' 09"	<1"	177' 46"
512	1' 11"	<1"	43' 21"
1024	1' 12"	<1"	12' 28"

It is worth observing that the computational time spent by the interpolation part is negligeable compared to the time spent by the probabilistic and the local solver part.

Example B. Consider now the more general problem

$$\frac{\partial u}{\partial t} = (1+x^2)\frac{\partial^2 u}{\partial x^2} + (1+y^3)\frac{\partial^2 u}{\partial y^2} + (\sin xe^y + 2)\frac{\partial u}{\partial x} + (\sin x\cos y + 2)\frac{\partial u}{\partial y} - u + \frac{1}{2}u^2 + \frac{1}{2}u^3$$
$$\text{in } \Omega = [-L, L] \times [-L, L], t > 0, \tag{52}$$

with $L = 1$ and boundary- and initial-conditions

$$u(x,y,t)|_{\partial\Omega} = 0, \quad u(x,y,0) = \cos^2\left(\frac{\pi x}{2L}\right)\cos^2\left(\frac{\pi y}{2L}\right). \tag{53}$$

No analytical solution is known for this problem, hence the numerical error made was controlled comparing the solution obtained with the PDD method with that given by a finite difference method with a very fine space-time mesh.

In Table 3, results similar to those of the Example A are shown, but now we used up to 1,024 processors. For this problem, the computational time spent by the probabilistic part is less than that spent in the previous case. The larger values of the diffusion coefficient allows now the random trees to reach the boundary faster than before. Therefore, the most part of the computational time is spent by the local solver, where the parallelization features of the algorithm can be exploited at best, the problem being fully uncoupled, at this point. Even though the problem of Example B is more complex than that of Example A, the PDD method behaves similarly, showing the excellent scalability properties of the algorithm for arbitrarily many processors.

Note that the algorithm scales according to a factor approximately equal to 4, when doubling the number of processors. However, a factor from 8 should be expected from (48), since $N/p \gg 1$. An explanation of this fact could be found in the size of the problem (hence in the computational load), which is not sufficiently large, and because the optimized LAPACK is speeding up considerably the LU factorization. Therefore, the dominant asymptotic behavior is governed, rather, by the second term in (48). To confirm that this is the case, new simulations for a larger computational load would be needed, such that now $N/p \gg 1/\alpha_{opt}$. The required computational resources, in terms of memory, however, largely exceed the available memory (the limits of the computational resources have already being reached). We can circumvent this problem increasing the bandwidth of the associated matrix, keeping almost constant the computational load. This can be done merely considering a rectangular domain, $\Omega = [-L_x, L_x] \times [-L_y, L_y]$, with $L_x \neq L_y$. In Table 4, the results corresponding to such a rectangular domain are shown. We used the parameters: $L_x = 204.8$, $L_y = 0.125$, $\Delta x = \Delta y = 5 \times 10^{-3}$, and $\Delta t = 10^{-3}$. Note that the scaling factor has changed significantly, reaching even the value 5, when passing from 128 to 256 processors. Increasing further the number of processors, implies reducing the scaling factor. This is in good agreement with the theoretical estimates in (48), since, asymptotically, the dominant term changes accordingly to the number of processors involved as well.

6. Summary

A new hybrid parallel numerical algorithm for solving nonlinear parabolic partial differential equations in any space dimension has been investigated, capable of exploiting the best features of two strategies: domain decomposition method, and the Monte Carlo method. A domain decomposition approach has been used to split the given space-time domain into as many subdomains as available processors. The solutions on the interfaces separating the subdomains, being unknown, are computed by interpolating on the nodal points where the solution is obtained probabilistically. This probabilistic computation consists of evaluating averages on suitably-generated random trees, which play a role similar to that of random paths in linear problems. In contrast to the classical deterministic method for solving partial differential equations, the probabilistic approach allows to compute the solution at single points internal to the domain, without the need for first generating a computational grid and solving the full problem. This fact is of paramount importance because, once the solution on the interfaces has been computed, the tasks of evaluating the solutions inside each subdomains turn out to be totally independent of one another, and thus can be assigned to an arbitrary number of processors without any intercommunication overhead. In principle, a drawback of the algorithm is the computational time spent in computing the solution at the nodal points by averaging over suitable random trees. Over large time intervals, the computational time spent for the probabilistic part of the algorithm becomes prohibitive, due to the appearance of random trees with large numbers of branches. However, in this paper it has been proved theoretically, and confirmed by numerical simulations, that the trees in question can be"pruned" in a suitable way, without appreciably losing accuracy. This strategy speeds up the probabilistic part considerably. Some numerical examples have been provided here, which show the excellent scalability properties of the PDD algorithm in large-scale simulations, using up to 1,024 processors on a high performance supercomputer. The performance of the PDD algorithm has been compared with that of other efficient, freely available parallel algorithms, showing a striking difference.

Acknowledgments

This work was supported, in part, by the Portuguese FCT, the Italy–Spain ‘‘Integrated Actions”, and the Italian GNFM-INdAM. The authors thankfully acknowledge computer resources, technical expertise, and assistance provided by the Barcelona Supercomputing Center - Centro Nacional the Supercomputación.

References

[1] J.A. Acebrón, M.P. Busico, P. Lanucara, R. Spigler, Domain decomposition solution of elliptic boundary-value problems, SIAM J. Sci. Comput. 27 (2) (2005) 440–457.
[2] J.A. Acebrón, M.P. Busico, P. Lanucara, R. Spigler, Probabilistically induced domain decomposition methods for elliptic boundary-value problems, J. Comput. Phys. 210 (2) (2005) 421–438.
[3] J.A. Acebrón, R. Spigler, Supercomputing applications to the numerical modeling of industrial and applied mathematics problems, J. Supercomput 40 (2007) 67–80.
[4] J.A. Acebrón, R. Spigler, A fully scalable parallel algorithm for solving elliptic partial differential equations, in: Lect. Notes in Comput. Sci., vol. 4641, 2007, pp. 727–736.
[5] J.A. Acebrón, Á. Rodríguez-Rozas, R. Spigler, Domain decomposition solution of nonlinear two-dimensional parabolic problems by random trees, submitted for publication.
[6] H.M. Antia, Numerical Methods for Scientists and Engineers, Tata McGraw-Hill, New Delhi, 1995.
[7] P. Arbenz, A. Cleary, J. Dongarra, M. Hegland, A comparison of parallel solvers for diagonally dominant and general narrow-banded linear systems, Parallel Distributed Comput. Pract. 2 (4) (1999) 385–400.
[8] P. Arbenz, A. Cleary, J. Dongarra, M. Hegland, A Comparison of Parallel Solvers for Diagonally Dominant and General Narrow-banded Linear Systems II, EuroPar'99 Parallel Processing, Springer, Berlin, 1999. pp. 1078–1087.
[9] P. Arbenz, M. Hegland, On the stable parallel solution of general narrow banded linear systems, in: P. Arbenz, M. Paprzycki, A. Sameh, V. Sarin (Eds.), High Performance Algorithms for Structured Matrix Problems, Nova Science Publishers, Commack, NY, 1998, pp. 47–73.
[10] L. Arnold, Stochastic Differential Equations: Theory and Applications, Wiley, New York, 1974.
[11] P. Baldi, Exact asymptotics for the probability of exit from a domain and applications to simulation, Ann. Prob. 23 (1995) 1644–1670.
[12] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK User's Guide, Society for Industrial and Applied Mathematics, SIAM, 1997.
[13] D. Blömker, M. Romito, R. Tribe, A probabilistic representation for the solutions to some non-linear PDEs using pruned branching trees, Ann. Inst. H. Poincaré Probab. Stat. 43 (2007) 175–192.
[14] F.M. Buchmann, W.P. Petersen, An Exit Probability Approach to Solve High Dimensional Dirichlet Problems, SIAM J. Sci. Stat. Comput. 28 (2006) 1153–1166.
[15] Tony F. Chan, Tarek P. Mathew, Domain Decomposition Algorithms, Acta Numerica, Cambridge University Press, Cambridge, 1994. pp. 61–143.
[16] C. DeBoor, A practical guide to splines, Springer, 1994.
[17] M. Freidlin, Functional Integration and Partial Differential Equations, Annals of Mathematics Studies, vol. 109, Princeton Univ. Press, Princeton, 1985.
[18] E. Floriani, R. Lima, R. Vilela Mendes, Poisson–Vlasov: stochastic representation and numerical codes, Eur. Phys. J. D 46 (2008) 295–302.
[19] G.A. Geist, Progress towards Petascale Virtual machines, in: J. Dongarra, D. Laforenza, S. Orlando (Eds.), Euro PVM/MPI 2003, Lecture Notes in Computer Science, Springer, Berlin, 2003, pp. 10–14.
[20] E. Gobet, Weak approximation of killed diffusion using Euler schemes, Stochast. Process. Appl. 87 (2000) 167–197.
[21] M. Hegland, Divide and conquer for the solution of banded linear systems of equations, in: Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing (PDP'96), January 24–26, 1996, p. 394.
[22] M.H. Kalos, P.A. Withlock, Monte Carlo Methods, Basics, vol. I, Wiley, New York, 1986.
[23] I. Karatzas, S.E. Shreve, Brownian Motion and Stochastic Calculus, second ed., Springer, Berlin, 1991.
[24] P.E. Kloeden, E. Platen, Numerical Solution of Stochastic Differential Equations, Springer, Berlin, 1992.
[25] Y. LeJan, A.-S. Sznitman, Stochastic cascades and 3-dimensional Navier–Stokes equations, Probab. Theory Related Fields 109 (1997) 343–366.
[26] R. Mannella, Absorbing boundaries and optimal stopping in a stochastic differential equation, Phys. Lett. A 254 (1999) 257–262.
[27] H.P. McKean, Application of Brownian motion to the equation of Kolmogorov–Petrovskii–Piskunov, Commun. Pure Appl. Math. 28 (1975) 323–331.
[28] M. Ossiander, A probabilistic representation of solutions of the incompressible Navier–Stokes equations in R^3, Probab. Theory Related Fields 133 (2005) 267–298.
[29] A. Quarteroni, A. Valli, Domain Decomposition Methods for Partial Differential Equations, Oxford Science Publications, Clarendon Press, Oxford, 1999.
[30] J.M. Ramirez, Multiplicative cascades applied to PDEs (two numerical examples), J. Comput. Phys. 214 (2006) 122–136.
[31] J. Seixas, R. Vilela Mendes, Large-deviation analysis of multiplicity fluctuations, Nuclear Phys. B 383 (1992) 622–642.

[32] W. Strittmatter, Numerical simulation of the mean first passage time, University Freiburg Report No. THEP 87/12, unpublished.
[33] E.V. Shikin, A.I. Plis, Handbook on Splines for the User, CRC-Press, 1995.
[34] R. Vilela Mendes, F. Cipriano, A stochastic representation for the Poisson–Vlasov equation, Commun. Nonlinear Sci. Numer. Simul. 13 (2008) 221–226.
[35] R. Vilela Mendes, J. Seixas, Large-deviation analysis of multiplicity fluctuations, Nuclear Phys. B 383 (1992) 622–642.
[36] E. Waymire, Probability and incompressible Navier–Stokes equations: a overview of some recent developments, Probab. Surv. 2 (2005) 1–32.