On the performance of a new parallel numerical algorithm for large-scale simulations of nonlinear partial differential equations

Juan A. Acebrón[1], Angel Rodríguez-Rozas[1], and Renato Spigler[2]

[1] Center for Mathematics and its Applications, Department of Mathematics, Instituto Superior Técnico Av. Rovisco Pais 1049-001 Lisboa, Portugal,
`juan.acebron@ist.utl.pt` (Juan A. Acebrón)
`angel.rodriguez@ist.utl.pt` (Angel Rodríguez-Rozas)
[2] Dipartimento di Matematica, Università "Roma Tre", Largo S.L. Murialdo 1, 00146 Rome, Italy,
`spigler@mat.uniroma3.it` (Renato Spigler)

Abstract. A new parallel numerical algorithm based on generating suitable *random trees* has been developed for solving nonlinear parabolic partial differential equations. This algorithm is suited for current high performance supercomputers, showing a remarkable performance and arbitrary scalability. While classical techniques based on a deterministic *domain decomposition* exhibits strong limitations when increasing the size of the problem (mainly due to the intercommunication overhead), probabilistic methods allow us to exploit massively parallel architectures since the problem can be fully decoupled. Some examples have been run on a high performance computer, being scalability and performance carefully analyzed. Large-scale simulations confirmed that computational time decreases proportionally to the cube of the number of processors, whereas memory reduces quadratically.

1 Introduction

An efficient design of numerical parallel algorithms for large-scale simulations becomes crucial when solving realistic problems arising from Science and Engineering. Most of them are based on *domain decomposition* (DD) techniques [12], since it has been shown to be specially suited for parallel computers. Nevertheless, classical approaches based on domain decomposition usually suffer from process intercommunication and synchronization, and consequently the scalability of the algorithm turns out to be seriously degraded. Moreover, an additional excess of computation is often introduced when designing the algorithms in order to diminish the effects of the two previous issues, and usually requires of some heuristics to attain the best performance. From the parallelism point of view, probabilistic methods based on Monte Carlo techniques offer a promising alternative to overcome these problems.

The possibility of using parallel computers to solve efficiently certain partial differential equations (PDEs) based on its probabilistic representation was

recently explored. In fact, in [1, 2], an algorithm for numerically solving linear two-dimensional boundary-value problems for elliptic PDEs, exploiting the probabilistic representation of solutions, were introduced for the first time. This consists in a hybrid algorithm, which combines the classical domain decomposition method [11, 14], and the probabilistic method, and was called "probabilistic domain decomposition method" (PDD for short). The probabilistic method was used merely to obtain 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. Known the solution at the interfaces, a full decoupling in arbitrarily many independent subdomains can be accomplished, and a classical local solver, arbitrarily chosen, used. This fact represents a definitely more advantageous circumstance, compared to what happens in any other existing deterministic domain decomposition method.

However, in contrast with the linear PDEs, probabilistic representation for nonlinear problems only exists in very particular cases. In [5], we extended the PDD method to treat nonlinear parabolic one-dimensional problems, while in [6] it was conveniently generalized to deal with arbitrary space dimensions.

In this paper, for the first time the performance of the PDD algorithm in terms of computational cost and the memory consumption will be carefully analyzed. To this purpose, large-scale simulations in a high performance supercomputer were runned, and the advantageous scalability properties of the algorithm verified. Since the local solver for the independent subproblems can be arbitrarily chosen, for this paper we chose a direct method based on LAPACK for solving the ensuing linear algebra problem. This is so because ScaLAPACK were selected to compare the performance of our PDD method, being ScaLAPACK a competitive, freely available and widely used parallel numerical library. Other alternatives based on iterative methods are worth to be investigated, and will be done in a future work.

In the following, we review briefly the mathematical fundamentals underlying the probabilistic representation of the solution of nonlinear parabolic PDEs. More precisely, for simplicity let consider the following initial-boundary value problem for a nonlinear two-dimensional parabolic PDE,

$$\frac{\partial u}{\partial t} = D_x(x,y)\frac{\partial^2 u}{\partial x^2} + D_y(x,y)\frac{\partial^2 u}{\partial y^2} + G_x(x,y)\frac{\partial u}{\partial x} + G_y(x,y)\frac{\partial u}{\partial y}$$

$$-c\,u + \sum_{i=2}^{m} \alpha_i u^i, \quad \text{in } \Omega = [-L,L]\times[-L,L], \ t > 0,$$

$$u(x,y,0) = f(x,y), \ u(x,y,t)|_{\mathbf{x}\in\partial\Omega} = g(x,y,t) \tag{1}$$

where $x, y \in \mathbf{R}^2$, $\alpha_i \in [0,1]$, $\sum_{i=2}^{m} \alpha_i = 1$, being $f(x,y)$ and $g(x,y,t)$, the initial condition and boundary data, respectively. The probabilistic representation for the solution of (1) was explicitly derived in [6] for the n-dimensional case, and requires generating suitable random trees with so many branches as the power of the nonlinearity found in (1). Such random trees play the role that

a random path plays in the linear case, (cf. Feynman-Kac formula for linear parabolic PDE). Mathematically, the solution of (1) for a purely initial value problem at $(\mathbf{x}, t)$ can be represented (see [13]) as follows

$$u(\mathbf{x}, t) = E_{\mathbf{x}}[\prod_{i=1}^{k(\omega)} f(\mathbf{x}_i(t, \omega))]. \tag{2}$$

Here $k(\omega)$ is the number of branches of the random tree starting at $\mathbf{x}$, and reaching the final time t. $\mathbf{x}_i(t, \omega)$ is the position of the ith branch of the random tree, corresponding to the stochastic process solution of the stochastic differential equation:

$$d\beta = b(x, y, t)\,dt + \sigma(x, y, t)\,dW(t), \tag{3}$$

where $W(t)$ represents the standard Brownian motion, and $b(x, y, t)$ and $\sigma(x, y, t)$, the corresponding drift and diffusion related to the coefficients of the elliptic operator in (1). The different possible realizations of the random trees are labeled by ω, and a suitable average $E_{\mathbf{x}}$ should be taken over all trees. In case of a initial-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.,

$$u(\mathbf{x}, t) = E_{\mathbf{x}} \left[\prod_{i=1}^{k(\omega)} \left\{ f(\mathbf{x}_i(t, \omega)) \mathbf{1}_{[S_i > \tau_\Omega]} + g(\beta_i(\tau_\Omega), \tau_\Omega) \mathbf{1}_{[S_i < \tau_\Omega]} \right\} \right], \tag{4}$$

where $\tau_{\partial\Omega}$ is the first exit time out of the domain of the stochastic process $\beta(\cdot)$; S_i a random time governed by the exponential distribution $p(S_i) = c \exp(-c\,S_i)$, and $\mathbf{1}_{[S_i < \tau_\Omega]}$ is the indicator function, being 1 or 0 depending whether $\tau_{\partial\Omega}$ is or is not greater than S_i.

For more mathematical details, we refer to the reader to [5] and [6]. In practice, the probabilistic representation in (2) works as follows: A stochastic process initially located in $(\mathbf{x}, 0)$ (root of the tree) is generated and evolves in time. Simultaneously, a random time S_1, picked up from the exponential probability density $p(S) = c \exp(-c\,S)$, is generated. If the random time S_1 is less than t, a number of branches corresponding to the power of the nonlinearity are created. This procedure is repeated successively until holds that $\sum_{i=1}^{n} S_i > t$. Whenever one of these possible branches reaches the final time, t, the initial value function, f, is evaluated at the position where the stochastic process associated to every branch was located. The solution is finally reconstructed multiplying all contributions coming from each branch. In practice, to compute the expected value in (4), only a finite number of random trees are generated, giving rise to the appearance of a statistical error. Therefore, to achieve a reasonable small error, a large number of random trees should be generated. More details on the numerical errors introduced by the probabilistic representation are found in [5] and [6]. For the purpose of illustration, in Fig. 1 a sketchy picture is shown. This corresponds to the case of a tree with two branches, one of them has reached the final time t, while the other hit the boundary.
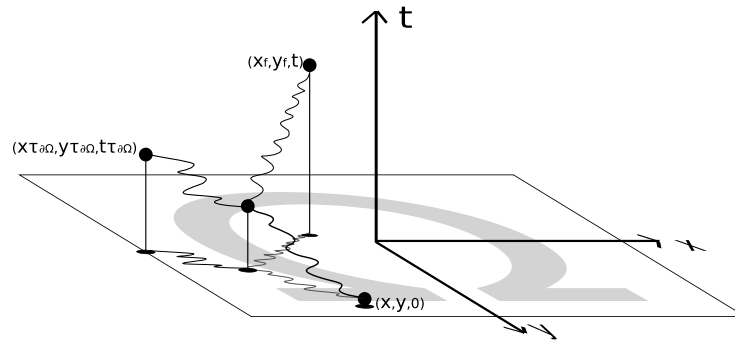
Fig. 1. A picture illustrating a typical random tree in 2D.

The paper is organized as follows. In Section 2 the PDD algorithm is described, showing their different components, and stressing how they can be parallelized in practice. Here we also discuss the important feature of being a natural fault-tolerant algorithm. In Section 3 some examples of nonlinear two-dimensional PDEs illustrate the PDD method, and a deep analysis of the computational cost and memory consumption is also undertaken. Finally, some conclusions are given.

2 The PDD method

As it was mentioned above, the core of algorithm requires evaluating the solution of (1) at given points using a probabilistic representation. Although this could be done at each given point of a given computational grid, in practice this is not often done, mainly because the computational time may be extremely high or even prohibitive due to the slow convergence of the Monte Carlo method. However, when the computational resources at hand are numerous, this procedure turns out to be feasible, and the following parallelization strategies can be adopted:

- *Parallelizing by splitting in independent sets of points.* Since the number of points where the solution is computed is larger than the number of processors p, the task to compute the solution at few of them can be assigned to different processors. This can be seen as a coarse-grain parallelization.
- *Parallelizing for each point.* Recall that for a very point, the probabilistic representation requires generating a rather large number of random trees to obtain a sufficiently small statistical error. Since the number of random trees turns out to be often much larger than p, the task to generate only a few of them can be accomplished by different processors. This corresponds to a mid-grain parallelization.
- *Parallelizing for a given realization.* Since every branch pertaining to a given tree is independent from each other, the task to compute only a few branches

can be assigned to different processors. This can be seen as a fine-grain parallelization, being by far the most involved, provided that a strong intercommunication and synchronization among the processors is required. However, this level of parallelization offers an important advantage, since it allows to balance easily the overall computational load.

Let now describe briefly the three parts the PDD algorithm is composed:

Probabilistic part. This is the first step to be carried out, and consists in computing the solution of the PDE at few suitable points by Monte Carlo, which is esentially a mesh-free method. In Fig. 2 a sketchy diagram is plotted, illustrating how the algorithm works in practice for a two-dimensional case. Here the solution is obtained probabilistically at few points pertaining to some "interfaces" conveniently chosen inside the space-time domain $D := \Omega \times [0,T]$, with $\Omega \subset \mathbf{R}^n$. Such interfaces divide the domain into p subdomains, Ω_i, from $i = 1, ..., p$, being assigned to different processors, $p_i, i = 1, ..., p$. For instance, in $\mathbf{R}^2$ this is done by splitting the domain in slices where the interfaces are parallel to the y-axis, as shown in Fig. 2. Since evaluating by Monte Carlo is computationally costly, only few points are chosen in each plane (y,t), and the task of computing them assigned to a different processor. In practice, every processor computes the interfacial values at the corresponding interface located to the right (see Fig. 2). Therefore, since there are $p - 1$ interfaces, the last processor has no task to accomplish, and thus it remains idle. For an increasing number of processors, this turns out to be irrelevant. Since each processor shares its interface with its immediately closer neighbor processor, the interfacial values must be transferred from the processor p_i to the processor p_{i+1}, establishing a next neighbor intercommunication among processors.
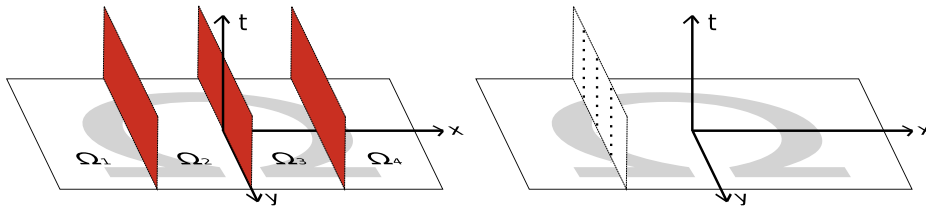
Fig. 2. A sketchy diagram illustrating the main steps of the algorithm in 2D: The figure on the left shows how the domain decomposition is done in practice. The figure on the right shows the points where the solution is computed probabilistically; these are used afterwards as nodal points for interpolation.

Interpolation. Once the solution has been computed at few points on each interface, a second step consists in interpolating on such points, being used as nodal points, thus obtaining continuous approximations of interfacial values of the solution. For that purpose, a tensor product interpolation based on cubic

spline [7] was used. The computational cost of the two dimensional case has been estimated from numerical experiments, and turns out to be negligible compared with the time spent in the other parts of the algorithm. The nodal points were uniformly distributed on each plane, and a not-a-knot condition was imposed

Local solver. The third and final step consists in computing the solution inside each subdomain, this task being assigned to different processors. This can be accomplished resorting to local solvers, which may use classical numerical schemes, such as finite differences or finite elements methods. Since the comparison with other freely available method was made using ScaLAPACK, LAPACK has been chosen for this purpose.

Fault tolerance-related issues

We stress that the algorithm turns out to be naturally fault-tolerant, being so in any of the parts it is composed. Here, we describe the peculiarities of such features:

- Probabilistic part. If a given processor p_i fails when computing the solution probabilistically at a given point, either any other processor may continue the pending task or, in case of no free processors at that time, this task may be postponed to be executed later. Another possibility rests in dropping such interface, merging the subdomains Ω_{i-1} and Ω_i in a new larger one, and reassigning it to the processor p_{i+1}.
- Interpolation. Similarly as in the previous case, when a processor fails, the subdomain corresponding to such a processor may be reassigned to a neighbor processor or the task may be postponed to be executed afterwards. Note that failure on a particular processor during the execution of this part affects merely this processor, since the remaining processors do not require any data from it.
- Local solver. Let suppose a failure occurs at a certain time $t = T_f$, when solving locally the PDE by a classical method. Then, the solution obtained at T_f may be used as a new initial data, solving the problem starting now from T_f rather than $t = 0$.

3   Results and performance

We now present some numerical examples for two-dimensional problems aiming to illustrate the performance of the PDD algorithm. All simulations were carried out on the MareNostrum Supercomputer located at the Barcelona Supercomputing Center, using up to 1,024 processors.

3.1   Computational cost and memory consumption

The space-time domain inside each subdomain corresponding to the PDD algorithm, as well as, the full domain to compare with the PDD, were discretized

using the implicit Crank-Nicolson finite difference method. On the various un-coupled subdomains obtained by the PDD algorithm we used LAPACK as a local solver. In [6], a comparison with the results obtaining with ScaLAPACK [10] was already carried out, showing serious limitations in terms of memory consumption. There, it was necessary to adjust carefully the free parameters of the test problems to be able to accomplish a fair comparison between both methods. A striking difference in the performance of both methods was shown, the performance of the PDD method being notably superior to ScaLAPACK.

Here we focus mainly on the strong scalability and performance of the PDD method. In the following, we keep fixed the space discretization of the two-dimensional finite difference computational mesh, being N_x and N_y the number of nodes in each dimension. The resulting matrix A of the linear algebra problem associated to discretization of the domain is known to be banded, with a bandwidth BW, say. Note that the bandwidth of the matrix A is N_x, being $A \in \mathbb{R}^{N_x N_y \times N_x N_y}$. Since we parallelized the PDD algorithm assigning the task of solving the local linear system pertaining to each subdomain to different processors, the bandwidth of associated matrix turns out to be smaller. The local solver for the PDD method, being based on LAPACK for solving banded linear systems, consists in a LU factorization followed by a forward/backward substitution. More specifically, the routines of LAPACK used were `dgbtrf/dgbtrs`. The memory consumption per processor, including an extra fill-in space, was also estimated in [6], and is given by

$$M_{P_{PDD}} = 3\frac{N_x^2 N_y}{p^2}. \tag{5}$$

Recall that the computational cost for the LU factorization is known to be of order of $N_x N_y BW^2$, while is $N_x N_y BW$ for the forward/backward substitution, $N_x N_y$ being the size of the square matrix and $BW = N_x/p$. Hence, the computational cost of the local solver for the PDD based on LAPACK is,

$$C_{PDD} \approx \alpha_{opt}(4\frac{N_x}{p} + 1)\ \frac{N_x}{p}\ \frac{N_x N_y}{p} + 3\frac{N_x^2 N_y}{p^2} + \frac{N_x N_y}{p}$$
$$= \frac{N_x^2}{p}\left(4\alpha_{opt}\frac{N_x N_y}{p^2} + (\alpha_{opt} + 3)\frac{N_y}{p}\right) + \frac{N_x N_y}{p} \tag{6}$$

see [8, 9, 6]. The last two terms are 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 using an optimized LAPACK library, after tuning conveniently the BLAS library for the specific hardware platform used. Typically, BLAS library can be easily tuned through the ATLAS package. To estimate experimentally the value of α_{opt} when runned the simulations at the MareNostrum Supercomputer, we have compiled BLAS with and without ATLAS, and it turns out to be approximately $\alpha_{opt} = 0.84$.

We stress that, when $N_x, N_y >> p$, the computational cost of the PDD method decreases as p^{-3} for large p, while for ScaLAPACK it has been reported in [8, 9] to decrease rather as p^{-1}.

3.2 Numerical results

In this subsection we consider two numerical test problems.
Example A. The problem is given by

$$\frac{\partial u}{\partial t} = (1 + x^4)\frac{\partial^2 u}{\partial x^2} + (1 + y^2 \sin^2(y))\frac{\partial^2 u}{\partial y^2} + (2 + \sin(x)e^y)\frac{\partial u}{\partial x} + (2 + x^2 \cos(y))\frac{\partial u}{\partial y}$$

$$-u + \frac{1}{2}u^2 + \frac{1}{2}u^3, \qquad \text{in } \Omega = [-L, L] \times [-L, L],\, 0 < t < T,$$

$$(7)$$

with boundary- and initial-conditions

$$u(x, y, t)|_{\partial\Omega} = 0, \quad u(x, y, 0) = \cos^2\left(\frac{\pi x}{2L}\right)\cos^2\left(\frac{\pi y}{2L}\right). \qquad (8)$$

No analytical solution is known for this problem, hence the numerical error
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.

Table 1. Example A: T_{TOTAL} denotes the computational time spent in seconds by
PDD. T_{MC}, and T_{INTERP} correspond to the time spent by the Monte Carlo part of
the algorithm, and the interpolation part, respectively. Finally, $Memory$ denotes the
total memory consumption.

No. Processors	T_{MC}	T_{INTERP}	$Memory$	T_{TOTAL}
128	445"	<1"	0.68 GBs	12976"
256	459"	<1"	0.17 GBs	3420"
512	463"	<1"	0.04 GBs	1166"
1024	461"	<1"	0.01 GBs	595"

In Fig. 3, the pointwise numerical error made when solving the problem of
Example A by PDD is shown. Here the value of the parameters were kept fixed
to $\Delta x = \Delta y = 10^{-2}$, $\Delta t = 10^{-3}$ and $L = 1$.
In Table 1 results from the Example A are shown, using up to 1,024 pro-
cessors. Now, the parameters chosen for the local solver were: $\Delta x = \Delta y = 2.5 \times 10^{-4}$, $T = 0.5$, $L = 1$ for the space domain, and $\Delta t = 10^{-3}$ for the time
domain. Note that the algorithm scales with the number of processors according
to a factor of approximately equal to 4 initially with $p = 128$, when doubling the
number of processors. This is so when the number of processors is not too large
$(p = 128)$, being the computational workload per processor sufficiently large for
this case. However, a factor from 8 should be expected theoretically from (6),
since $N_x/p, N_y/p >> 1$. 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 (6). To confirm that this is the case, new simulations for a larger computational load would be needed, such that now $N_x/p, N_y/p >> 1/\alpha_{opt}$. The required computational resources, in terms of memory, however, largely exceed the available memory (the limits of the computational resources at hand have already being reached). A simple way to circumvent this problem consists in increasing the bandwidth of the associated matrix, keeping almost constant the computational load. This can be done merely considering a rectangular domain instead, $\Omega = [-L_x, L_x] \times [-L_y, L_y]$, with $L_x > L_y$. Other alternatives based on adopting an heterogeneous computational grid, or even using higher order numerical schemes are equally possible, and will be analyzed elsewhere.

From the results above, it can be observed that increasing the size of the problem by raising the number of nodes N_x along the $x-$dimension, results in a computational cost growing cubically with N_x, whereas the memory consumption grows quadratically. Rather, the effect on the y-dimension is more contained, since both the computational cost and the memory consumption increase linearly with the number of nodes N_y. Therefore, to increase the computational workload appreciably it is more convenient to increase the size of the problem along the $x-$dimension. In fact, this is successfully illustrated in the next example.

Example B. Consider the following problem

$$\frac{\partial u}{\partial t} = (1+x^2)\frac{\partial^2 u}{\partial x^2} + (1+y^3)\frac{\partial^2 u}{\partial y^2} + (2+\sin(x)\,e^y)\frac{\partial u}{\partial x} + (2+\sin(x)\cos(y))\frac{\partial u}{\partial y}$$

$$-u + \frac{1}{2}u^2 + \frac{1}{2}u^3, \qquad \text{in } \Omega = [-L_x, L_x] \times [-L_y, L_y], \, 0 < t < T, \qquad (9)$$

and boundary- and initial-conditions

$$u(x,y,t)|_{\partial\Omega} = 0, \quad u(x,y,0) = \cos^2\left(\frac{\pi x}{2L_x}\right)\cos^2\left(\frac{\pi y}{2L_y}\right). \qquad (10)$$

In Table 2, the results corresponding to such a rectangular domain are shown. We used the parameters: $L_x = 204.8$, $L_y = 0.125$, $\Delta x = \Delta y = 5 \times 10^{-3}$, $T = 0.5$, 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 (6), since, asymptotically, the dominant term changes accordingly to the number of processors involved as well.

4 Conclusions and future work

The performance of a new parallel numerical algorithm for solving nonlinear parabolic partial differential equations based on a probabilistic method, has been investigated. Such a method allows to decouple the full problem into several independent subproblems, the computational cost being dramatically alleviated.

Table 2. Example B: T_{TOTAL} denotes the computational time spent in seconds by PDD. T_{MC}, and T_{INTERP} correspond to the time spent by the Monte Carlo part of the algorithm, and the interpolation part, respectively. Finally, $Memory$ denotes the total memory consumption.

No. Processors	T_{MC}	T_{INTERP}	*Memory*	T_{TOTAL}
128	67"	<1"	1.79 GBs	54235"
256	69"	<1"	0.45 GBs	10666"
512	71"	<1"	0.11 GBs	2601"
1024	72"	<1"	0.03 GBs	748"

Many different sources of parallelism for this algorithm can be explored efficiently, since intercommunication overhead among processors is almost nonexistent. Moreover, the algorithm appears to be fault-tolerant in a natural way. While the computational cost for traditional schemes scales linearly with the number of processors, the PDD algorithm scales quadratically or even cubically, provided that the workload per processor is sufficiently large. Some numerical examples are given, showing the excellent scalability properties of the PDD algorithm in large-scale simulations, using up to 1,024 processors on a high performance supercomputer. Summarizing, the proposed algorithm appears to be a promising alternative to the traditional parallel schemes to solve partial differential equations. Such an algorithm is characterized by desirable features, such as low intercommunication overhead, and fault-tolerance, allowing to exploit at best massively parallel supercomputers as well as Grid computing.

Acknowledgments

References

1. Acebrón, J.A., Busico, M.P., Lanucara, P., and Spigler, R., *Domain decomposition solution of elliptic boundary-value problems*, SIAM J. Sci. Comput. **27**, No. 2 (2005), 440-457.
2. Acebrón, J.A., Busico, M.P., Lanucara, P., and Spigler, R., *Probabilistically induced domain decomposition methods for elliptic boundary-value problems*, J. Comput. Phys., **210**, No. 2 (2005), 421-438.
3. Acebrón, J.A., and Spigler, R., *Supercomputing applications to the numerical modeling of industrial and applied mathematics problems*, J. Supercomputing, 40 (2007) 67-80.
4. Acebrón, J.A., and Spigler, R., *A fully scalable parallel algorithm for solving elliptic partial differential equations*, Lect. Notes in Comput. Sci. **4641** (2007) 727-736.

5. Acebrón, J.A., Rodríguez-Rozas, A., and Spigler, R., *Efficient parallel solution of nonlinear parabolic partial differential equations by a probabilistic domain decomposition*, (2009), submitted.

6. Acebrón, J.A., Rodríguez-Rozas, A., and Spigler, R., *Domain decomposition solution of nonlinear two-dimensional parabolic problems by random trees*, J. Comput. Phys., in press (2009).

7. Antia, H.M., *Numerical methods for scientists and engineers*, Tata McGraw-Hill, New Delhi, 1995.

8. Arbenz, P., Cleary, A., Dongarra, J., and Hegland, M., *A comparison of parallel solvers for diagonally dominant and general narrow-banded linear systems*, Parallel and Distributed Computing Practices, **2**, No. 4 (1999), 385-400.

9. Arbenz, P., Cleary, A., Dongarra, J., and Hegland, M., *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.

10. Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C., *ScaLAPACK User's Guide*, Society for Industrial and Applied Mathematics, SIAM, 1997.

11. Chan, Tony F., and Mathew, Tarek P., *Domain decomposition algorithms*, Acta Numerica (1994), 61-143 [Cambridge University Press, Cambridge, 1994].

12. Keyes, D. E., *Domain Decomposition Methods in the Mainstream of Computational Science*, Proceedings of the 14th International Conference on Domain Decomposition Methods, UNAM Press, Mexico City, (2003), pp. 79-93.

13. McKean, H.P., *Application of Brownian motion to the equation of Kolmogorov-Petrovskii-Piskunov*, Comm. on Pure and Appl. Math., **28** (1975), 323-331.

14. Quarteroni, A., and Valli, A., *Domain Decomposition Methods for Partial Differential Equations*, Oxford Science Publications, Clarendon Press, Oxford, 1999.

12

Fig. 3. Pointwise numerical error for the solution of problem "Example A", at $t = 0.5$, using two *processors*. Note that there is only one interface located at $x = 0$.