A fully scalable parallel algorithm for solving elliptic partial differential equations

Juan A. Acebrón[1] and Renato Spigler[2]

[1] Departament d'Enginyeria Informàtica i Matemàtiques,
Universitat Rovira i Virgili, 43007 Tarragona, Spain,
juan.acebron@urv.cat
[2] Dipartimento di Matematica, Università "Roma Tre", 1, Largo S.L. Murialdo,
00146 Rome, Italy
spigler@mat.uniroma3.it

Abstract. A comparison is made between the probabilistic domain decomposition (DD) method and a certain deterministic DD method for solving linear elliptic boundary-value problems. Since in the deterministic approach the CPU time is affected by intercommunications among the processors, it turns out that the probabilistic method performs better, especially when the number of subdomains (hence, of processors) is increased. This fact is clearly illustrated by some examples. The probabilistic DD algorithm has been implemented in an MPI environment, in order to exploit distributed computer architectures. Scalability and fault-tolerance of the probabilistic DD algorithm are emphasized.

1 Introduction and generalities

Domain decomposition is considered as one of the most natural ways to decouple boundary-value (BV) problems for partial differential equations (PDEs) into subproblems, in order to take advantage of parallel computer architectures, thus allowing for high-performance scientific computing of large-scale problems. The seminal idea of DD can be traced back to the 1870 work of H. A. Schwarz, and consists of splitting the given domain into a number of subdomains, then assigning the task of the numerical solution on each separate subdomain to as many separate processors. A major problem, however, is represented by the need of having the solution on the interfaces, internal to the domain, which divide it into subdomains, while solving BV problems for PDEs is global in character. This means that the solution cannot be obtained even at a single point inside the domain before solving the entire problem. Consequently, some iterative procedures are required, across the chosen (or prescribed) interfaces, in order to determine approximate values of the sought solution inside the original domain. Both, overlapping (Schwarz-type) and not overlapping domains have been considered in the literature so far [13]. In both cases, some additional numerical work is required in advance to decouple the problem into subproblems, and it is unclear whether algorithms based on such strategies might be *scalable* as the number of the subdomains (hence, of the processors) increases unboundedly.

In the classical (deterministic) DD method, an important issue is represented by the condition number inherent to the iterative algorithms adopted to precompute interfacial values, see [13]. In fact, the operator governing such an iterative procedure is typically ill-conditioned. Preconditioning is therefore essential, and it is necessary to construct optimal and efficient preconditioners. Optimality means that their spectral condition number (i.e., the ratio between maximum and minimum eigenvalue) is bounded uniformly with respect to the mesh size, h, and to the average diameter, say H, of the subdomains. It is known from the literature that in Schwarz-type domain decomposition methods, that is those with overlapping, the aforementioned condition numbers are actually independent of both, h and H, provided that the overlapping is sufficiently wide. Optimal bounds were indeed found for this case, see [5, 7].

A method of completely new type, based on a probabilistically induced domain decomposition (PDD), has been recently proposed which avoids the several problems inherent to the aforementioned traditional approach to DD [1, 2, 4]. Moreover, the method seems to be specially suited for heterogeneous computing, due to its low communication overhead, and since it is fault-tolerant.

Nowadays, it seems that, using parallel computers with up to a few hundreds of processors in problems with up to few millions of variables, the total computational cost is dominated by that spent by the local solvers. However, machines working in the petaflops regime and endowed with hundreds of thousands or even millions of processors are planned for the near future, and taking full advantage from massive parallel computing would be highly desirable. Indeed, the IBM Blue Gene is expected to break the petaflops barrier within the 2007. With such machines, the issue of *scalability* remains open, at least in some cases. As it was pointed out in [11], Schwarz-type DD methods are not truly scalable, at least in the theoretical sense, since their parallel efficiency in solving elliptic problems is subject to degradation as the number of processors, p, goes to infinity, indeed when p starts being over the thousands. It seems however that things go a little better, in practice, for a number of reasons, as described in [11].

Besides, the possibility of failure of even few processors (even of only one!) is very likely to occur frequently [9]. Therefore, algorithms which are scalable and *fault-tolerant* at the same time would (and will) be extremely important, if not mandatory.

Briefly, the idea of the PDD algorithm consists of generating first the values of the solution at few points inside the domain. Then, an interpolation on such nodes is constructed, and finally the traces of the solution on the interfaces are obtained. This allows to fully decompose the domain into a number of subdomains, and at the same time this procedure seems to be free of the previous drawbacks. In fact, decoupling is complete and no conditioning problem does exist concerning interfacial iteration problems. In [1, 2], it was shown that scalability is attained with respect to an arbitrary number of subdomains or processors, and the algorithm is naturally fault-tolerant. The latter property rests on two ingredients, one due to the intrinsic parallelizability of the Monte Carlo methods, the other to the full decoupling into subdomains that can be

realized. As for the scalability, it was shown in [1] that the speedup S_p achieved with p processors, assumed to act independently from each other on p subdomains, scales as $S_p \sim c\sqrt{p}$ as $p \to \infty$, c being a constant. The ideal (theoretical) speedup, $S_p \sim c\,p$, cannot be attained since some (nonnegligeable) time should be spent to compute the values of the solution by Monte Carlo at few points. Here the speedup is defined as the ratio T_1/T_p, where T_1 is the time spent to solve sequentially the given problem on the full domain, while T_p is the time spent for solving the same problem in parallel with p processors.

In the paper [4] the essentials of the PDD method developed in [1, 2] have been described, and some additional examples have been presented. In all such papers, however, the code implementing the PDD algorithm was written in OpenMP environment. The numerical results worked out in this paper are based, instead, on a code written in MPI environment. This has been done in order to fully exploit distributed computer architectures. The main purpose of this paper is to compare the performance of the PDD and of the deterministic DD algorithms implemented in MPI environment.

2 The algorithm

We confine our discussion to the case of the Dirichlet problem for a linear elliptic equation in two dimensions, i.e.,

$$Lu - c(x,y)u = f(x,y), \quad (x,y) \in \Omega \subset \mathbf{R}^2, \qquad u|_{\partial\Omega} = g(x,y), \tag{1}$$

where $L := \sum_{i,j=1}^{2} a_{ij}(x,y)\partial_i\partial_j + \sum_{i=1}^{2} b_i(x,y)\partial_i$ is a linear elliptic operator with smooth coefficients, $c(x,y) \geq 0$, the boundary $\partial\Omega$ of the domain Ω also smooth, as well as the boundary data, g, and the source term, f. The basic idea is to generate only few values of solution, u, by a probabilistic method, all being based on the Monte Carlo method [6].

In some sense, this approach allows to obtain the solution at any point internal to Ω without solving the entire problem in advance. This can be realized by means of the the probabilistic representation of the solution,

$$u(x,y) = E^L_{x,y}\left[g(\beta(\tau_{\partial\Omega}))e^{-\int_0^{\tau_{\partial\Omega}} c(\beta(s))\,ds} - \int_0^{\tau_{\partial\Omega}} f(\beta(t))\,e^{-\int_0^t c(\beta(s))\,ds}\,dt\right] \tag{2}$$

[8], where $\beta(t)$ is the two-dimensional stochastic process associated to the elliptic operator L, which solves the system of two Ito-type stochastic differential equations (SDEs)

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

Here $W(t)$ represents the two-dimensional standard Brownian motion (also called Wiener process), and $\tau_{\partial\Omega}$ is the first passage (or hitting) time of the path $\beta(t)$ started at the point (x,y) to $\partial\Omega$. When the operator L is the Laplace operator, Δ, the stochastic process $\beta(t)$ reduces to the standard two dimensional Brownian motion. The drift vector, $b = b(x,y)$ in (3), is the same appearing in the operator

L, that is $b = (b_1, b_2)^T$, while the diffusion matrix, $\sigma = \sigma(x, y)$, is related to the coefficients a_{ij} by the relation $\sigma\sigma^T = a \equiv (a_{i,j})_{i,j=1,2}$.

We use the representation formula in (2) to obtain few values of the solution, at some points inside the domain Ω. The expected value is then approximated by an arithmetic mean (known to provide its best estimator) over N realizations of the process β, at the price of a (statistical) error of order of $N^{-1/2}$. The main catch of using a Monte Carlo method rests on this fact, which entails a rather poor accuracy, unless N is taken extremely large. Even for N not very large, the computational cost is high, but in the PDD algorithm we limit its use to very few points. The points where the solution is computed by Monte Carlo simulations are then used as nodes for interpolation. Such interpolation allows to obtain boundary values of solution on certain interfaces internal to the domain, and hence to fully decouple the original problem into subproblems, see Fig. 1. We always use Chebyshev interpolation.

The PDD algorithm can be briefly described as follows:

1. Compute only *few* interfacial values by Monte Carlo simulations.
2. Interpolate on the corresponding nodes to obtain boundary values for the subdomains.
3. Compute the solution to the original problem in each subdomain by standard methods (e.g., finite differences or finite elements).

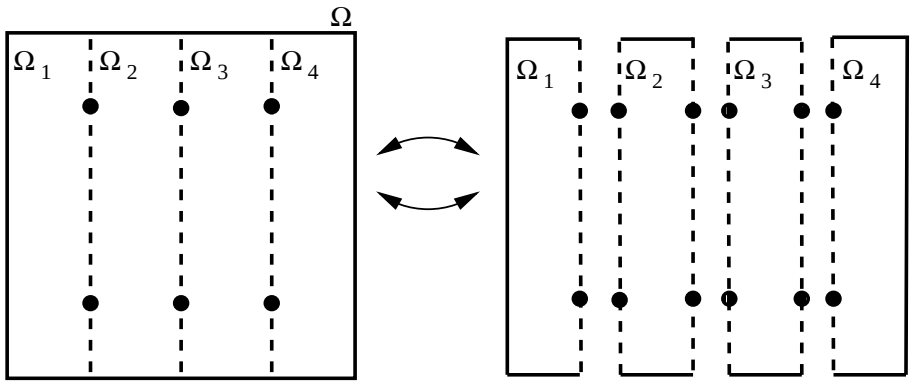

Fig. 1. Sketchy diagram which shows how the PDD algorithm works.

The Monte Carlo approach is trivially parallelizable, since each of the N problems above can be runned independently of the others. Stated in such terms, clearly, the method appears to be scalable as well. Moreover, it is also naturally fault-tolerant. In fact, assuming that a fraction, n, of the N processors being used fails, meaningful results will be obtained from the remaining $N - n$ processors,

and it will suffice just to ignore what could have come from the n processors which have failed. The final error will be slightly worse, of the order of $\mathcal{O}((N-n)^{-1/2})$. Note that

$$(N-n)^{-1/2} \approx N^{-1/2}\left(1+\frac{1}{2}\frac{n}{N}\right), \quad \text{for } n \ll N, \tag{4}$$

that is $1.005\,N^{-1/2}$ for a small fraction $n/N = 1/100$ (1%) of failing processors.

Even though the probabilistic algorithm we derived and considered here is scalable, naturally fault tolerant, and well suited to grid and heterogeneous computing, it suffers for some weakness, due to the inherently poor accuracy of all Monte Carlo methods. A considerable improvement, however, can be achieved using sequences of "quasi-random numbers" [6, 12] instead of sequences of pseudorandom numbers. The pseudorandom numbers are those numbers obtained in practice, when we try to generate truly random numbers, hence are approximately characterized by a statistical distribution. The quasi-random numbers, instead, are deterministic uniformly distributed numbers. Using the latter allows to obtain an error (now deterministic) of order of $\mathcal{O}(N^{-1}\log^{d^*-1} N)$, d^* representing a certain "effective" space dimension. It was shown in [3] that the underlying system of SDEs can indeed be solved numerically in a very efficient way. However, a reordering strategy is required at each step to break somehow the inherent correlations of the sequence of numbers. This makes it difficult to parallelize the algorithm, and raises some doubts on whether using quasi-random numbers can be useful in practice. But if this would be possible, the failure of $n \ll N$ processors would imply the slightly larger error

$$\begin{aligned}(N-n)^{-1}\log(N-n) = N^{-1}\left(1-\frac{n}{N}\right)^{-1}\left[\log N + \log\left(1-\frac{n}{N}\right)\right] \\ \approx (1+\alpha)N^{-1}\left[\log N - \alpha\right].\end{aligned} \tag{5}$$

Here we assumed $d^* = 2$ and set $\alpha := n/N$. With $\alpha = 0.01$, and considering that N will be at least of the order of the thousands, hence $\log N \gg \alpha$ (e. g., $\log N = 3$ against $\alpha = 0.01$), we obtain

$$(N-n)^{-1}\log(N-n) \approx (1+\alpha)N^{-1}\log N = 1.01\,N^{-1}\log N. \tag{6}$$

Moreover, due to the full uncoupling among the various subdomains, the PDD algorithm exhibits fault-tolerance also when those processors working on small fractions of subdomains fail. In this case, one can fully neglect (in the first instance) the corresponding output, without the need of resorting to restart procedures.

The various sources of numerical error which affect the evaluation of $u(x, y)$ by means of (2) include, besides the finite sample size mentioned above, (*i*) the truncation error made in the numerical solution of the SDEs in (3), not to mention the round-off errors, (*ii*) the uncertainty in estimating first exit times (hitting times), and (*iii*) the numerical quadrature errors in (2). The latter is missing whenever the potential term, $c(x, y)$, and the source, $f(x, y)$, in (1) are

identically zero. In particular, estimating accurately first exit times and first exit points (which are also needed when $c(x, y)$ and $f(x, y)$ do not vanish identically), has been often overlooked in the literature. An efficient way to locate accurately first exit times is adopting exponential timesteppings, see [10]. This choice allows in fact to use an explicit analytic form of the hitting probability. All these sources of error have been analyzed in [1, 2], and we shall not do it here again.

3 Numerical examples

Some numerical examples are provided here to compare the performance of the PDD algorithm described in §1 with that obtained by means of certain deterministic DD algorithms. Despite the fact that "pivotal" values generated by the Monte Carlo method are poorly accurate and the Chebyshev interpolation adds some further error, the total error inside of each subdomain is estimated by the boundary errors. This is due to the maximum principle, and the error decays rapidly going inside.

In [1, 2, 4], a comparison was made only with "parallel finite differences", hence a comparison with a true deterministic DD method was not made there. Moreover, the code was implemented in OpenMP, and runned in a shared memory computer architecture. In addition to the fact that the PDD method outperforms the deterministic DD method, the former wins regarding to the issues of scalability and fault-tolerance. These properties, not easily achieved by the DD deterministic methods, should instead be considered at the present time if one wants to exploit the forthcoming massively parallel supercomputers, equipped with hundreds of thousands or even millions of processors. The numerical examples presented below concern, for the purpose of illustration, partial differential equations on rather elementary domains, indeed, the unit square. Monte Carlo methods, however, are expected to be useful even on domains having highly complex geometries. Here we only stress that our probabilistic DD approach can be easily applied, for instance, to polygonal domains with an arbitrary number of sides. In fact, one of the most delicate points, which contributes significantly to the overall numerical error, is given by the need of evaluating accurately the first exit times and points, see [1, 2]. While such a task may be challenging when the boundaries are very irregular, it can be handled easily when the boundary is piecewise linear. We do not enter into details in this paper.

Another related issue is that of having one or more rather general interfaces, inside the domain. Their shape may be prescribed, due to geometrical or physical reasons. The idea is to "approximate" every (sufficiently smooth) interface by a piecewise linear one, i.e., by a polygonal line, evaluating numerically on it few values of the sought solution, to be used for interpolation. This can be accomplished computing the solution by Monte Carlo methods at some more points on such a line. These points should include the endpoints of each segment of the polygonal, say the $k-2$ internal points for a polygonal line of k segments, and perhaps another point on each segment. The overall cost may therefore increase, and load balancing become a little harder.

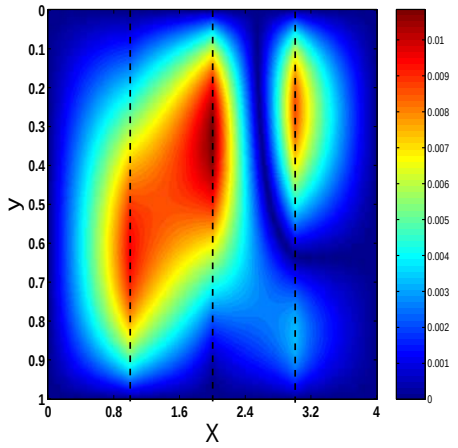

Fig. 2. Example 1. Pointwise numerical error in the PDD algorithm for $p = 4$. Parameters are $N = 10^5$, $\Delta x = \Delta y = 1.25 \times 10^{-3}$, $\lambda = 10^2$.

Finally, we stress that, due to the full decoupling that the PDD method realizes, arbitrary and possible different algorithms can be implemented to solve the problem on each subdomain. In particular, serial solvers can be adopted, taking into account that one will face smaller-size problems on each subdomain. However, the point in serious applications is not that to cope with small-size problems, but, rather, to be able to solve problems that are much bigger.

A code in an MPI environment has been implemented and runned on the MareNostrum supercomputer located at the Barcelona Supercomputing Center (BSC). Below, in order to compare the relative performance of the PDD method against some deterministic method, we solved the same examples in both ways. The chosen deterministic algorithm is extracted from the numerical package pARMS, having chosen the overlapping Schwarz method with a FGMRES iterative method preconditioned with ILUT as local solver, see [14]. We chose the package pARMS for its wide and reliable usage, though better codes may exist for a comparison. The values of the parameters were chosen as follows: For the outer iterations, the Krylov subspace dimension was 20 and the tolerance 10^{-5}, while for the ILUT preconditioner the dropping threshold was 10^{-3} and the amount of fill in any row 20. The inner iterations were set equal to zero. To make the comparison meaningful, we discretized the local problems within the PDD algorithm by finite differences, and solved the ensuing linear algebraic system by the same FGMRES iterative solver preconditioned with ILUT. Here are our numerical examples.

Example 1. Consider the following Dirichlet problem,

$$u_{xx} + u_{yy} = 0 \qquad \text{in } \Omega := (0, p) \times (0, 1), \tag{7}$$

$$u(x,y)|_{\partial\Omega} = g(x,y), \tag{8}$$

where $g(x,y) := \left[\left(x^2 - y^2\right)/p^2\right]_{\partial\Omega}$, for the Laplace equation in two dimensions. The solution of such a problem is explicitly known, and is $u(x,y) = (x^2 - y^2)/p^2$ in $\overline{\Omega} = [0,p] \times [0,1]$. Note that the domain was scaled along the x dimension proportionally to the number of processors, p, involved. This has been done in order to keep constant the computational load per processor, being the space discretization fixed to $\Delta x = \Delta y = 1.25 \times 10^{-3}$.

In Fig. 2, the pointwise numerical error made in the PDD method is depicted in a contourplot. Here only two nodes on each interface have been used. Note that the maximum error made in each subdomain is indeed attained on the corresponding boundary. The exponential timestepping used to solve the underlying SDEs (3) was characterized by $\lambda := \langle \Delta t \rangle^{-1}$, Δt being the random exponentially distributed time step used in solving the SDEs, and the bracket denoting its average. Note that maximum error is of order 10^{-2}, which corresponds mainly to the statistical error obtained from the Monte Carlo method at the nodal points. Increasing the accuracy can be attained by increasing the sample size, or resorting to sequences of "quasi-random" numbers keeping fixed the sample size, see [2].

Table 1. CPU time in seconds

Processors	*DD*	*PDD*
4	110.49	84.05
8	132.23	84.12
16	163.81	90.84
32	192.22	90.54
64	335.20	88.89
128	14184.35	102.22
256	*NA*	132.15
512	*NA*	129.35
1024	*NA*	148.07

Table 1 shows the CPU time spent in solving the present problem by the two algorithms. Clearly, the PDD outperforms the DD method for any number of processors. This fact becomes more pronounced when the number of processors increases. This behavior can be explained by the high intercommunication overhead existing among processors, which affects strongly the deterministic algorithm. Note that the CPU time for the PDD method remains bounded when the number of processors grows. Testing scalability for larger number of processors (starting for $p = 256$) have been accomplished only for the PDD, because CPU time for DD increases unboundedly. For this reason, in table 1 CPU times are Not Available (NA).

Example 2. The Dirichlet problem

$$u_{xx} + (6x^2 + 1)u_{yy} = 0 \qquad \text{in } \Omega = (0, p) \times (0, 1), \tag{9}$$

with the boundary data

$$u(x, y)|_{\partial\Omega} = \left[(x^4 + x^2 - y^2)/2(p^4 + p^2)\right]_{\partial\Omega}, \tag{10}$$

has the solution $u(x, y) = (x^4 + x^2 - y^2)/2(p^4 + p^2)$. Again, the CPU times are reported in Table 2. The same comments made in the previous example can be repeated here.

Table 2. CPU time in seconds

Processors	*DD*	*PDD*
4	115.94	69.38
8	110.80	70.06
16	113.41	70.61
32	133.09	70.12
64	255.07	72.19
128	25827.83	75.73
256	*NA*	67.08
512	*NA*	65.67
1024	*NA*	64.12

4 Conclusions

A probabilistic method, to accomplish domain decomposition for the numerical solution of linear elliptic boundary-value problems in two dimensions, has been described. The solution is generated by Monte Carlo simulations to solve the associated stochastic differential equations only at very few points inside the domain. A Chebyshev interpolation using such points as nodes is then constructed, and a full splitting into several subdomains, to be handled by separate processors acting concurrently, is made.

A comparison with a deterministic DD algorithm has been made here for the first time. This has been done in an MPI environment. Working in an MPI environment also allows to test the effect of processor intercommunications which beset all deterministic DD algorithms. Besides the competitive results observed in the numerical examples, the PDD method is expected to be competitive concerning scalability and fault-tolerance. These are key issues if one intends to run codes on machines working with hundreds of thousands of processors or more. Finally, we believe the PDD method could be applied, and likely more advantageously, in three or more dimensions. In practice, some more work is needed,

especially concerning the important issue of accurately compute first exit times of the trajectories of the underlying stochastic processes from high-dimensional domains.

Acknowledgements.

This work was completed during a visit of J.A.A. at the Department of Mathematics, University "Roma Tre", and was supported, in part, by the GNFM of the Italian INdAM. J.A.A. also acknowledges support from the Ministerio de Ciencia y Tecnología (MEC) through the Ramón y Cajal programme. The authors thankfully acknowledge the computer resources, technical expertise and assistance provided by the Barcelona Supercomputing Center-Centro Nacional de Supercomputación.

References

1. Acebrón, J.A., Busico, M.P., Lanucara, P., and Spigler, R.,"Domain decomposition solution of elliptic boundary-value problems via Monte Carlo and quasi-Monte Carlo methods", SIAM J. Sci. Comput., **27**, 440–457 (2005).
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**, 421–438 (2005).
3. Acebrón, J.A., and Spigler, R., "Fast simulations of stochastic dynamical systems", J. Comput. Phys., **208**, 106–115 (2005).
4. Acebrón, J.A., and Spigler, R., "A new probabilistic approach to the domain decomposition method", Lect. Notes in Comput. Sci. and Eng., Vol. 55, 475–480 (2007)
5. Brenner, S.C., " Lower bounds of two-level additive Schwarz preconditioners with small overlap", SIAM J. Numer. Anal., **21**, 1657–1669 (2000).
6. Caflisch, R.E., "Monte Carlo and quasi Monte Carlo methods", Acta Numerica. Cambridge University Press, 1–49. (1998).
7. Dryja, M., and Widlund, O.B., "Domain decomposition algorithms with small overlap", SIAM J. Sci. Comput., **15**, 604–620 (1994).
8. Freidlin, M.: Functional integration and partial differential equations. Annals of Mathematics Studies no. 109, Princeton Univ. Press (1985).
9. Geist, G.A., "Progress towards Petascale Virtual Machines", Lecture Notes in Computer Science, **2840**, 10–14 (2003).
10. Jansons, K.M., and Lythe, G.D., " Exponential timestepping with boundary test for stochastic differential equations", SIAM J. Sci. Comput., **24** 1809–1822 (2003).
11. Keyes, D.E., " How scalable is domain decomposition in practice?" in the *Eleventh International Conference on Domain Decomposition Methods* (London, 1998), 286-297 (electronic), DDM.org, Augsburg, (1999).
12. Niederreiter, H.: Random number generation and quasi Monte-Carlo methods. SIAM (1992).
13. Quarteroni, A., and Valli, A.: Domain decomposition methods for partial differential equations. Oxford Science Publications, Clarendon Press (1999).
14. Li, Z., Saad, Y., and Sosonkina, M., "pARMS: a parallel version of the algebraic recursive multilevel solver", Numerical Linear Algebra with Applications, **10**, 485–509 (2003).