Journal of Computational Physics **157,** 419–440 (2000)
doi:10.1006/jcph.1999.6338, available online at http://www.idealibrary.com on IDEAL®

Tensorial Basis Spline Collocation Method for Poisson's Equation

Laurent Plagne* and Jean-Yves Berthou†

**Département de Recherche Fondamentale sur la Matière Condensée, CEA-Grenoble, 17 Rue des Martyrs, 38054 Grenoble Cedex 9, France; and †EDF-DER/IMA/MMN/ISA, 1, Av. du General de Gaulle, 92141 Clamart Cedex, France*
E-mail: plagne@ilt.fhg.de

Received June 29, 1998; revised April 15, 1999

This paper aims to describe the tensorial basis spline collocation method applied to Poisson's equation. In the case of a localized 3D charge distribution in vacuum, this direct method based on a tensorial decomposition of the differential operator is shown to be competitive with both iterative BSCM and FFT-based methods. We emphasize the $O(h^4)$ and $O(h^6)$ convergence of TBSCM for cubic and quintic splines, respectively. We describe the implementation of this method on a distributed memory parallel machine. Performance measurements on a Cray T3E are reported. Our code exhibits high performance and good scalability: As an example, a 27 Gflops performance is obtained when solving Poisson's equation on a 256^3 non-uniform 3D Cartesian mesh by using 128 T3E-750 processors. This represents 215 Mflops per processors.

Key Words: Poisson solver; Vlasov equation; orthogonal spline collocation; tensor-product; multipolar expansion; direct method; 3D Cartesian non-uniform grids; parallel implementations.

1. INTRODUCTION

Elliptic equations occur in a large class of physical problems and the computational cost of their solution is usually a major factor in computer simulation code design. Depending on the physical problem to be solved, one must choose very carefully a numerical method from among the large number of methods available for this class of differential equations. In practice, the main issue for 3D problems is to achieve the shortest time of calculation for a given accuracy. This calculation time, considered as the main feature of the numerical method, is a function of three parameters: the grid size, the algorithmic cost, and the efficiency of the implemented code. The grid size (the number of grid cells) needed to reach a given accuracy depends on the order of the method (e.g., the order of a finite difference

0021-9991/00 $35.00

scheme) and on the suitability of the grid to the physical problem. The algorithmic cost, which is the number of operations required to perform the calculation as a function of the grid size, is obviously another essential feature. The efficiency of the implemented code depends mainly on the optimization level attainable with the compiler used on sequential computers, together with the suitability of the method for a parallel implementation on parallel computers. Obviously, in order to judge the merit of a proposed method, practical parameters for a physical problem (e.g., the required accuracy) must be clearly defined. This paper aims to describe a numerical method as practically as possible, that is to say, keeping in mind the overriding importance of the calculation time.

The method presented here has been designed to solve Poisson's equation for a localized 3D charge distribution in vacuum in the context of cluster physics [7, 14]. In this case, one can obtain the boundary conditions using a multipolar expansion, which requires, for high accuracies, the use of very large grids compared to the spatial extent of the charge density. Non-uniform grids are highly desirable, since they allow an accurate meshing of the charged region and at the same time, a large spatial extent of the grid using a relatively low number of grid cells. The non-uniform Cartesian grids which are used make this method an intermediate case between FFT-based methods which are very fast but use uniform grids not well suited to a multipolar expansion, and finite element methods, which are slower but very efficient for handling problems with a complex geometrical structure. Another essential feature of the method is the use of a cubic or quintic spline basis. The principle is to look for an approximate solution of the differential equation as an expansion on a spline basis. Solving the original differential equation is then equivalent to solving a linear system $AX = B$ where the matrix A is huge and sparse in the 3D case. The basis spline collocation method is described in detail in the paper [18]. The authors deal not only with elliptic equations but also with eigenvalue problems (e.g., the Schroedinger equation) and indicate an iterative method to solve the resulting linear systems. However, a direct (i.e., non-iterative) algorithm can be used in the particular case of elliptic equations on Cartesian grids by maintaining a tensorial structure throughout the calculations.

Part 2 describes the derivation of the method. In order to introduce all the spline-related notations, the case of one-dimensional differential equations is discussed in Subsection 2.1. The 3D case is treated in Subsection 2.2 where all tensorial notations are introduced. The computational cost $O(N^4)$ of the method is calculated in Subsection 2.3, allowing one to compare this direct method with iterative methods. Subsection 2.4 gives a detailed treatment of boundary conditions. In Subsection 2.5 we generalize the method to the Helmholtz equation and to the use of quintic splines. The $O(h^4)$ and $O(h^6)$ orders of TBSCM for the cubic and quintic cases respectively are emphasized in Subsection 2.6. Subsection 2.7 gives a comparison between TBSCM and FFT-based methods $O(N^3 \ln(N))$.

Section 3 describes the parallel implementation of TBSCM on Cray distributed memory machines. Subsection 3.1 introduces the different target machines. The mixed HPF-MPI parallel implementation of the computational kernel is detailed in Subsection 3.2. Performances and scaling of the whole code are emphasized in Subsection 3.3.

2. TENSORIAL BASIS SPLINE COLLOCATION METHOD

2.1. *Spline Basis and the One-Dimensional Case*

2.1.1. Cubic splines. Because several definitions can be found for a cubic spline [3], we first give the definition to be used in this paper. Let us first introduce a grid composed of $N_g + 1$ points $\{x_a\}/x_0 < x_1 < \cdots < x_a < \cdots < x_{N_g}$ on which are defined $2N_g + 2$

third-order piecewise polynomial functions (cubic splines): $\{S_i(x)\}/i = 0 \ldots 2N_g + 1$. One can define uniquely the two kinds of splines, namely odd and even splines, by giving the following conditions at the grid points:

$$\begin{aligned} S_{2a}(x_a) &= S'_{2a+1}(x_a) = 1 \\ S'_{2a}(x_a) &= S_{2a+1}(x_a) = 0 \\ S_{2a}(x) &= S_{2a+1}(x) = 0 \qquad \forall x \in [x_0, x_{a-1}] \cup \left[x_{a+1}, x_{N_g}\right]. \end{aligned}$$

These conditions lead to the explicit formulas

$$\begin{aligned} S_{2a}(x) &= \begin{cases} 3\left(\frac{x - x_{a-1}}{x_a - x_{a-1}}\right)^2 - 2\left(\frac{x - x_{a-1}}{x_a - x_{a-1}}\right)^3 & \forall x \in [x_{a-1}, x_a] \\ 3\left(\frac{x_{a+1} - x}{x_{a+1} - x_a}\right)^2 - 2\left(\frac{x_{a+1} - x}{x_{a+1} - x_a}\right)^3 & \forall x \in [x_a, x_{a+1}] \\ 0 & \forall x \in [x_0, x_{a-1}] \cup \left[x_{a+1}, x_{N_g}\right] \end{cases} \\ S_{2a+1}(x) &= \begin{cases} (x_a - x_{a-1})\left[\left(\frac{x - x_{a-1}}{x_a - x_{a-1}}\right)^2 - \left(\frac{x - x_{a-1}}{x_a - x_{a-1}}\right)^3\right] & \forall x \in [x_{a-1}, x_a] \\ (x_{a+1} - x_a)\left[\left(\frac{x_{a+1} - x}{x_{a+1} - x_a}\right)^2 - \left(\frac{x_{a+1} - x}{x_{a+1} - x_a}\right)^3\right] & \forall x \in [x_a, x_{a+1}] \\ 0 & \forall x \in [x_0, x_{a-1}] \cup \left[x_{a+1}, x_{N_g}\right]. \end{cases} \end{aligned} \tag{1}$$

Figure 1 shows odd and even cubic splines.

A function f expanded on this basis is differentiable with a continuous first derivative on the grid,

$$f(x) = \sum_{i=0}^{2N_g+1} c_i S_i(x).$$

Because there is only one non-zero spline and one non-zero differentiated spline at a grid point x_a, $f(x_a)$ and $f'(x_a)$ are given simply by

$$\begin{aligned} f(x_a) &= c_{2a} S_{2a}(x_a) = c_{2a} \\ f'(x_a) &= c_{2a+1} S'_{2a+1}(x_a) = c_{2a+1}. \end{aligned}$$

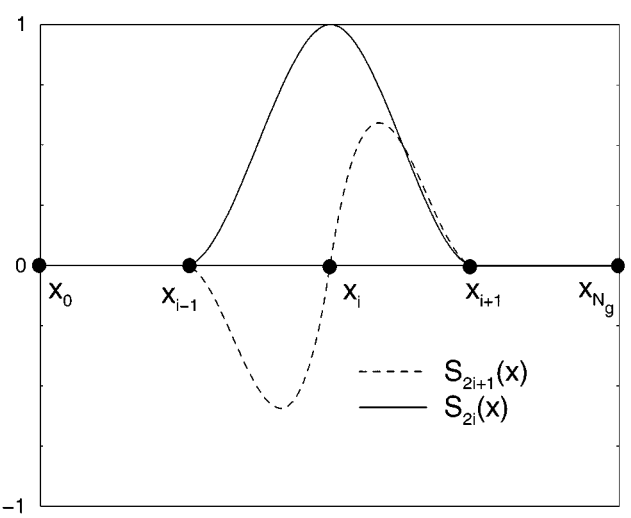

FIG. 1. Odd and even cubic splines.

2.1.2. One-dimensional equation. In order to describe the basis spline collocation method, we shall treat a simple differential equation in one dimension. Let us consider the differential equation

$$\frac{\partial^2 V(x)}{\partial x^2} = \rho(x). \tag{2}$$

Boundary conditions $V(x_0) = V_0$ and $V(x_{N_g}) = V_{N_g}$ are known. We are looking for an approximate solution $\phi(x) \simeq V(x)$ as an expansion on the spline basis,

$$\phi(x) = \sum_{i=0}^{2N_g+1} c_i S_i(x). \tag{3}$$

Hence, the problem is now to determine the $\{c_i\}$ coefficients. Obviously, two coefficients are already known: $c_0 = V_0$ and $c_{2N_g} = V_{N_g}$. In order to determine the $2N_g$ remaining unknown coefficients $\{c_i\}$ one has to define $2N_g$ equations.

2.1.3. Collocation points. Let us now introduce the so-called $2N_g$ collocation points $\{\bar{x}_\alpha\}$ chosen to be the Gauss points on each grid step $[x_a, x_{a+1}]$,

$$\begin{aligned} \bar{x}_{2a+1} &= \frac{(x_{a+1} + x_a)}{2} - \frac{(x_{a+1} - x_a)}{2\sqrt{3}} \\ \bar{x}_{2a+2} &= \frac{(x_{a+1} + x_a)}{2} + \frac{(x_{a+1} - x_a)}{2\sqrt{3}}. \end{aligned} \tag{4}$$

For convenience, we also define $\bar{x}_0 = x_0$ and $x_{2N_g+1} = x_{N_g}$. We thus obtain, by applying the differential equation (2) at each collocation point, the $2N_g$ equation,

$$\sum_{i=0}^{2N_g+1} c_i S_i''(\bar{x}_\alpha) = \rho(\bar{x}_\alpha) \qquad \forall \alpha \in [1, 2N_g]. \tag{5}$$

We will continue to use Greek indexes for collocation points (i.e., real space) while Roman indexes will be used for splines.

The derivatives S_i' and S_i'' are determined analytically from the polynomial expression (1).

2.1.4. Matrix notations. The discretization of the problem has been achieved by the use of a cubic spline basis. It is now useful to introduce a matrix notation. We define two matrices S and S'' $[2N_g + 2)^2]$ and vectors ϕ and C $[2N_g + 2]$ as

$$\forall \alpha, i \in [0, 2N_g + 1] \begin{cases} S_{\alpha i} = S_i(\bar{x}_\alpha) \\ S''_{\alpha i} = S_i''(\bar{x}_\alpha) \\ \phi_\alpha = \phi_i(\bar{x}_\alpha) \\ C_i = c_i. \end{cases}$$

It is now possible to rewrite these equations in a condensed form,

$$\begin{aligned} \phi(\bar{x}_\alpha) &= \sum_{i=0}^{2N_g+1} c_i S_i(\bar{x}_\alpha) \qquad \forall \alpha \in [0, 2N_g + 1] \\ &\Leftrightarrow \phi = SC. \end{aligned} \tag{6}$$

We introduce the operator $D = S''S^{-1}$, which satisfies $D\phi = S''S^{-1}SC = S''C$. We can rewrite Eq. (5) in the form

$$[D\phi]_\alpha = \sum_{i=0}^{2N_g+1} S''_{\alpha i} C_i = \sum_{i=0}^{2N_g+1} c_i S''_i(\bar{x}_\alpha) \qquad \forall \alpha \in [0, 2N_g+1]$$
$$= \rho(\bar{x}_\alpha) \qquad \forall \alpha \in [1, 2N_g]. \tag{7}$$

Boundary conditions must be separated from the unknown variables c_i (i.e., in the rhs). This is achieved by splitting this sum in two terms,

$$[D\phi]_\alpha = \sum_{\alpha'=0}^{2N_g+1} D_{\alpha\alpha'}\phi_{\alpha'} = \sum_{\alpha'=1}^{2N_g} D_{\alpha\alpha'}\phi_{\alpha'} + \sum_{A=0;2N_g+1} D_{\alpha A}\phi_A. \tag{8}$$

By combining Eqs. (7) and (8), the discretized differential equation can be rewritten

$$\sum_{\alpha'=1}^{2N_g} D_{\alpha\alpha'}\phi_{\alpha'} = \rho(\bar{x}_\alpha) - \sum_{A=0;2N_g+1} D_{\alpha A}\phi_A \qquad \forall \alpha \in [1, 2N_g]. \tag{9}$$

In order to rewrite this equation in a matrix form some new matrices and vectors must be defined. We define the sub-matrices and sub-vectors of dimensions $[(2N_g)^2]$ and $[2N_g]$, respectively,

$$\forall \alpha, \alpha' \in [1, 2N_g] \begin{cases} \tilde{D}_{\alpha\alpha'} = D_{\alpha\alpha'} \\ \tilde{\phi}_\alpha = \phi_\alpha \\ \tilde{\rho}_\alpha = \rho_\alpha - (D_{\alpha 0}\phi_0 + D_{\alpha 2N_g+1}\phi_{2N_g+1}). \end{cases}$$

The final expression of the original differential equation is therefore

$$\tilde{D}\tilde{\phi} = \tilde{\rho}. \tag{10}$$

At this point, one can easily solve Eq. (10) by performing the inversion of the square matrix $\tilde{D}$. Then, it is straightforward to get $\tilde{\phi} = \tilde{D}^{-1}\tilde{\rho}$. The $\tilde{\phi}$ vector contains the approximate solution at all collocation points. It should be noticed that several methods (e.g., finite difference method) stop at this point. In the BSCM case, the interpolation needed to get the approximate solution between collocation points is indeed natural. The use of spline basis in Eq. (3) transforms a continuous problem into a discrete one and can obviously be used to recover a continnous approximate solution. By building ϕ from $\tilde{\phi}$ and the boundary conditions, and by inverting Eq. (6), one can obtain $C = S^{-1}\phi$ and use Eq. (3) to get a continuous approximate solution known everywhere inside the grid.

2.2. *Three-Dimensional Poisson's Equation*

Subsections 2.2.1 and 2.2.2 show that the procedure for the three-dimensional case is a straightforward generalization of the one-dimensional case. One can easily obtain an equation similar to Eq. (10) that can be solved by using an iterative scheme, as proposed in [18]. Subsection 2.2.3 proposes a direct method based on tensorial decomposition which differs from the iterative BSCM treatment.

Let us first consider Poisson's equation using Hartree units. $V(\mathbf{r})$ is the electrostatic potential due to $\rho(\mathbf{r})$, a known localized charge density,

$$\nabla^2 V(x, y, z) = -4\pi\rho(x, y, z). \tag{11}$$

For each dimension, a grid (which can be different in size and location) must be chosen: $\{x_a\}$, $\{y_b\}$, $\{z_c\}$. To simplify the notations, the grid sizes are chosen to be equal, $Nx = Ny = Nz = N_g$. From these three grids, three $[2N_g + 2]$ collocation grids are defined (Eq. (4)): $\{\bar{x}_\alpha\}$, $\{\bar{y}_\beta\}$, $\{\bar{z}_\gamma\}$. Here again, the aim is to find an approximate solution to Eq. (11) defined on a 3D cubic spline basis,

$$\phi(x, y, z) = \sum_{ijk=0}^{2N_g+1} c_{ijk} S_{Xi}(x) S_{Yj}(y) S_{Zk}(z). \tag{12}$$

The discrete problem is now to determine the $(2N_g + 2)^3$ unknown variables c_{ijk} by applying the differential equation at the $(2N_g)^3$ inner collocation points and by using the $(2N_g + 2)^3 - (2N_g)^3$ boundary conditions. We now assume that the values of the potential at the surface of the 3D rectangular grid are known,

$$\phi_{\alpha\beta\gamma} \qquad \text{for } \alpha \text{ or } \beta \text{ or } \gamma = 0 \text{ or } 2N_g + 1. \tag{13}$$

We define, as we have done for the 1D case, the matrices $[(2N_g + 2)^2]$ S_X, S_Y, S_Z, S''_X, S''_Y, S''_Z, D_X, D_Y, D_Z and the sub-matrices $[(2N_g)^2]$ $\tilde{D}_X$, $\tilde{D}_Y$, $\tilde{D}_Z$. The sub-vector $\tilde{\rho}$ includes once again the boundary conditions,

$$\tilde{\rho}_{\alpha\beta\gamma} = 4\pi\rho(\bar{x}_\alpha, \bar{y}_\beta, \bar{z}_\gamma) - \sum_{A=0;2N_g+1} D_{X\alpha A}\phi_{A\beta\gamma} - \sum_{B=0;2N_g+1} D_{Y\beta B}\phi_{\alpha B\gamma} - \sum_{C=0;2N_g+1} D_{Z\gamma C}\phi_{\alpha\beta C}.$$

In order to be able to write down the equation corresponding to Eq. (10), we have to introduce new notations.

2.2.1. Tensorial notations. Let us consider three square matrices $[(2N_g + 2)^2]$ relative to each dimension: A_X, A_Y, A_Z. We introduce a tensorial operator A_{XYZ} with $[(2N_g + 2)^6]$ elements as the tensorial product of the three A operators, $A_{XYZ} = A_X \otimes A_Y \otimes A_Z$. Each element of this three-dimensional operator is defined by

$$[A_{XYZ}]_{\alpha\beta\gamma,ijk} = A_{X\alpha i} A_{Y\beta j} A_{Z\gamma k}.$$

Two three-dimensional operators can be combined by a tensor–tensor product,

$$A_{XYZ} = B_{XYZ} C_{XYZ}$$
$$\Leftrightarrow [A_{XYZ}]_{\alpha\beta\gamma,ijk} = \sum_{abc} [B_{XYZ}]_{\alpha\beta\gamma,abc} [C_{XYZ}]_{abc,ijk}.$$

The tensor-vector product between a three-dimensional operator A_{XYZ} and a three-dimensional vector U_{XYZ} with $[(2N_g+2)^3]$ elements is also defined,

$$\begin{aligned} V_{XYZ} &= A_{XYZ}U_{XYZ} \\ &\Leftrightarrow [V_{XYZ}]_{\alpha\beta\gamma} = \sum_{abc} [A_{XYZ}]_{\alpha\beta\gamma,abc}[U_{XYZ}]_{abc}. \end{aligned}$$

Introdcing the three-dimensional identity operator $I_{XYZ} = I_X \otimes I_Y \otimes I_Z$, one can easily verify the properties

$$\begin{cases} (A_X \otimes B_Y \otimes C_Z)^{-1} = A_X^{-1} \otimes B_Y^{-1} \otimes C_Z^{-1} \\ (A_{XYZ}B_{XYZ})^{-1} = B_{XYZ}^{-1}A_{XYZ}^{-1}. \end{cases} \tag{14}$$

2.2.2. Discrete Poisson's equation on a tensorial form. By using these newly introduced definitions, one can now write Eq. (12) applied at the collocation points,

$$\phi_{XYZ} = [S_X \otimes S_Y \otimes S_Z]C_{XYZ}. \tag{15}$$

It is now possible to write down in a condensed form Poisson's equation corresponding to Eq. (10) in the one-dimensional case. We first build the Cartesian Laplacian operator $\tilde{\nabla}^2_{XYZ}$,

$$\tilde{\nabla}^2_{XYZ} = [\tilde{D}_X \otimes \tilde{I}_Y \otimes \tilde{I}_Z + \tilde{I}_X \otimes \tilde{D}_Y \otimes \tilde{I}_Z + \tilde{I}_X \otimes \tilde{I}_Y \otimes \tilde{D}_Z]. \tag{16}$$

Poisson's equation can finally be written

$$\tilde{\nabla}^2_{XYZ}\tilde{\phi}_{XYZ} = \tilde{\rho}_{XYZ}. \tag{17}$$

At this point, the one-dimensional problem was over and the solution $\tilde{\phi}$ could be obtained by $\tilde{\phi} = \tilde{D}^{-1}\tilde{\rho}$. Similarly, one could easily transform Eq. (17) into a standard matrix equation by introducing super-indexes,

$$\begin{aligned} I &= i + 2j(N_g+1) + 4k(N_g+1)^2 \\ \Gamma &= \alpha + 2\beta(N_g+1) + 4\gamma(N_g+1)^2. \end{aligned}$$

Here, I corresponds uniquely to a set (i, j, k) and Γ to a set (α, β, γ). Using this one-to-one mapping, one can transform the tensorial operator $\tilde{\nabla}^2_{XYZ}$, and the tensorial vectors $\tilde{\phi}_{XYZ}$, $\tilde{\rho}_{XYZ}$, and C_{XYZ}, into standard matrix and vectors,

$$\begin{aligned} [\tilde{\nabla}^2]_{\Gamma\Gamma'} &= \left[\tilde{\nabla}^2_{XYZ}\right]_{\alpha\beta\gamma,\alpha'\beta'\gamma'} \\ [\tilde{\phi}]_\Gamma &= [\tilde{\phi}_{XYZ}]_{\alpha\beta\gamma} \\ [\tilde{\rho}]_\Gamma &= [\tilde{\rho}_{XYZ}]_{\alpha\beta\gamma} \\ [C]_I &= [C_{XYZ}]_{ijk}. \end{aligned}$$

One can rewrite Eq. (17) as a matrix equation equivalent to the 1D Eq. (10),

$$\tilde{\nabla}^2\tilde{\phi} = \tilde{\rho}. \tag{18}$$

The dimension of the matrix $\tilde{\nabla}^2$ is $(2N_g)^6$. For a typical number of collocation points, $2N_g = 100$ this leads to 10^{12} matrix elements. This matrix is fortunately extremely sparse. From the definition of the cubic splines one can see that there are only 4 non-zero splines at a given collocation point in one dimension. In the 3D case this leads to 64 non-zero elements per matrix row and to $64(2N_g)^3$ non-zero elements for the matrix $\tilde{\nabla}^2$. However, one cannot perform the inversion of this matrix because the inverse of a sparse matrix is not sparse and contains 10^{12} non-zero elements, which is impossible to store. The usual way to solve this kind of problem is to calculate $\tilde{\phi}$ using an iterative method. Basically, one tries a guessed solution $\tilde{\phi}^i$, then performs the product $\tilde{\rho}^i = \tilde{\nabla}^2\tilde{\phi}^i$ and uses the difference $\tilde{\rho} - \tilde{\rho}^i$ to calclate an improved guess $\tilde{\phi}^{i+1}$. This scheme is repeated iteratively until a given accuracy is reached. It should be noticed that the initial guess $\tilde{\phi}^i$ must be chosen "good enough" to ensure the convergence of the iterative scheme.

At each step, the matrix-vector product $\tilde{\nabla}^2\tilde{\phi}^i$ requires $128(2N_g)^3$ operations. However, because of the tensorial structure of $\tilde{\nabla}^2_{XYZ}$, a direct (i.e., non-iterative) method to solve Eq. (17) is available.

2.2.3. Tensorial decomposition. This direct method relies on the fact that the inverse tensorial operator $\tilde{\nabla}^{2^{-1}}_{XYZ}$ can be calculated as a function of small 1D matrices and a 3D operator made as simple as possible (diagonal). The first step is to diagonalize $\tilde{D}_X$, $\tilde{D}_Y$, $\tilde{D}_Z$,

$$\tilde{D}_X = \tilde{M}_X\tilde{\Pi}_X\tilde{M}_X^{-1}; \qquad \tilde{D}_Y = \tilde{M}_Y\tilde{\Pi}_Y\tilde{M}_Y^{-1}; \qquad \tilde{D}_Z = \tilde{M}_Z\tilde{\Pi}_Z\tilde{M}_Z^{-1}, \tag{19}$$

where $\tilde{\Pi}_X$, $\tilde{\Pi}_Y$, $\tilde{\Pi}_Z$ are diagonal matrices,

$$\begin{aligned} [\tilde{\Pi}_X]_{\alpha\alpha'} &= \delta_{\alpha\alpha'} l_{x\alpha} \\ [\tilde{\Pi}_Y]_{\beta\beta'} &= \delta_{\beta\beta'} l_{y\beta} \\ [\tilde{\Pi}_Z]_{\gamma\gamma'} &= \delta_{\gamma\gamma'} l_{z\gamma}. \end{aligned} \tag{20}$$

Since the operators $\tilde{D}_X$, $\tilde{D}_Y$, $\tilde{D}_Z$ are not symmetrical, this diagonalization can lead to complex matrices $\tilde{M}$ and $\tilde{\Pi}$. In the case of cubic splines on uniform Cartesian grids, these matrices have been proved to be real [2]. Since the complex case should be a straightforward generalization, we assume these matrices to be real in the following. Using this transformation, one can rewrite Eq. (16) as

$$\tilde{\nabla}^2_{XYZ} = \tilde{M}_X\tilde{\Pi}_X\tilde{M}_X^{-1} \otimes \tilde{I}_Y \otimes \tilde{I}_Z + \tilde{I}_X \otimes \tilde{M}_Y\tilde{\Pi}_Y\tilde{M}_Y^{-1} \otimes \tilde{I}_Z + \tilde{I}_X \otimes \tilde{I}_Y \otimes \tilde{M}_Z\tilde{\Pi}_Z\tilde{M}_Z^{-1}.$$

Using an obvious property of the identity operator, $\tilde{I}_X = \tilde{M}_X\tilde{I}_X\tilde{M}_X^{-1}$, we obtain

$$\tilde{\nabla}^2_{XYZ} = (\tilde{M}_X \otimes \tilde{M}_Y \otimes \tilde{M}_Z)\tilde{P}_{XYZ}\left(\tilde{M}_X^{-1} \otimes \tilde{M}_Y^{-1} \otimes \tilde{M}_Z^{-1}\right)$$

with

$$\tilde{P}_{XYZ} = \tilde{\Pi}_X \otimes \tilde{I}_Y \otimes \tilde{I}_Z + \tilde{I}_X \otimes \tilde{\Pi}_Y \otimes \tilde{I}_Z + \tilde{I}_X \otimes \tilde{I}_Y \otimes \tilde{\Pi}_Z.$$

The main point is that $\tilde{P}_{XYZ}$ is diagnal for all dimensions allowing one to calculate its inverse,

$$\left[\tilde{P}^{-1}_{XYZ}\right]_{\alpha\beta\gamma,\alpha'\beta'\gamma'} = \frac{\delta_{\alpha\alpha'}\delta_{\beta\beta'}\delta_{\gamma\gamma'}}{l_{x\alpha} + l_{y\beta} + l_{z\gamma}}.$$

Using the properties (14) one can finally write the inverse operator $\tilde{\nabla}^{2^{-1}}_{XYZ}$,

$$\tilde{\nabla}^{2^{-1}}_{XYZ} = (\tilde{M}_X \otimes \tilde{M}_Y \otimes \tilde{M}_Z)\tilde{P}^{-1}_{XYZ}\left(\tilde{M}_X^{-1} \otimes \tilde{M}_Y^{-1} \otimes \tilde{M}_Z^{-1}\right). \tag{21}$$

This direct approach to performing the inversion of a tensorial operator was derived for the first time in the context of the finite difference schemes in [13].

$\tilde{\phi}_{XYZ}$ is obtained by performing the tensor-vector product,

$$\tilde{\phi}_{XYZ} = \tilde{\nabla}^{2^{-1}}_{XYZ}\rho_{XYZ}. \tag{22}$$

Then, one can rebuild ϕ_{XYZ} and obtain C_{XYZ} using Eq. (15),

$$C_{XYZ} = \left(S_X^{-1} \otimes S_Y^{-1} \otimes S_Z^{-1}\right)\phi_{XYZ}. \tag{23}$$

Using Eq. (12), the approximate solution and its two first derivatives are known everywhere inside the mesh.

2.3. *Algorithmic Cost; Comparison with Iterative Methods*

Most of the mathematical papers dealing with elliptic partial differential equations give a detailed account of the number of floating-point operations required for a given algorithm. In many of them, a 2D square mesh is used to describe a method. Practically, in the 3D case, calculations on 1D matrices (e.g., $\tilde{D}_X = \tilde{M}_X\tilde{\Pi}_X\tilde{M}_X^{-1}$) have a computational cost very low compared to the cost of the 3D calculations involved in the method. The gap between the cost of 1D and 3D calculations is even bigger when Poisson's equation is to be solved many times on the same grid because all the 1D matrices (S_x, $\tilde{D}_x$, $\tilde{M}_x$, ...) are built once, whereas 3D calculations like (22) have to be done each time. By far the most expensive steps of the method are the two tensor-vector products in (22). As an example, the tensor-vector product (23) can be split in three operations (for convenience we define $N = 2N_g + 2$ to be the number of collocation points),

$$\begin{aligned}[C_{XYZ}]_{ijk} &= \sum_{\alpha\beta\gamma=0}^{N-1} \left[S_X^{-1}\right]_{i\alpha}\left[S_Y^{-1}\right]_{j\beta}\left[S_Z^{-1}\right]_{k\gamma}[\phi_{XYZ}]_{\alpha\beta\gamma} \\ &= \sum_{\alpha\beta=0}^{N-1} \left[S_X^{-1}\right]_{i\alpha}\left[S_Y^{-1}\right]_{j\beta}\left(\sum_{\gamma=0}^{N-1}\left[S_Z^{-1}\right]_{k\gamma}[\phi_{XYZ}]_{\alpha\beta\gamma}\right).\end{aligned}$$

We calculate successively the two intermediate vectors C'_{XYZ} and C''_{XYZ} and obtain finally C_{XYZ},

$$[C'_{XYZ}]_{\alpha\beta k} = \sum_{\gamma=0}^{N-1}\left[S_Z^{-1}\right]_{k\gamma}[\phi_{XYZ}]_{\alpha\beta\gamma}$$

$$[C''_{XYZ}]_{\alpha jk} = \sum_{\beta=0}^{N-1}\left[S_Y^{-1}\right]_{j\beta}[C'_{XYZ}]_{\alpha\beta k}$$

$$[C_{XYZ}]_{ijk} = \sum_{\alpha=0}^{N-1}\left[S_X^{-1}\right]_{i\alpha}[C''_{XYZ}]_{\alpha jk}.$$

One can easily count from these three equations that the tensor-vector product costs $3N^4$

real multiplications and $3N^4$ real additions. Equation (22) requires 2 tensor-vector products. The total algorithmic cost of the method is about $12N^4$ floating point operations ($18N^4$ if the interpolation (23) is done). The parallel implementation and optimization of this tensor-vector product is detailed in Section 3.

Each step of the corresponding iterative method costs $128N^3$ (see Subsection 2.2). Assuming 100 to be a typical value for N, it is straightforward to see that an iterative method should not exceed 10 iterations to remain competitive with TBSCM. For this size of linear system, 10 iterations are very few, even if a suitable preconditioner is used, and one would usually expect at least 50 iterations to reach an acceptable solution. For a given N, the maximum number of iterations leading to a competitive iterative scheme is

$$\frac{12N^4}{128N^3} = \frac{12}{128}N.$$

In addition, the use of a direct method allows us to avoid handling possible convergence problems of iterative methods.

2.4. *B-Spline Extras; Boundary Conditions*

2.4.1. Differentiation and integration procedure. It has been emphasized in the description of the 1D case that the natural interpolation (23) is not necessary when the electrostatic potential is only required at the collocation points. However, in many physical applications, one has to evaluate the potential and the electric field **E** everywhere inside the mesh. This is obviously the case when trajectories of charged particles are needed. In order to evaluate this field one has to differentiate the potential given by Eq. (12). For the X component of this field this leads to

$$\frac{\partial \phi}{\partial x}(x, y, z) = \sum_{ijk=0}^{N} c_{ijk} S'_{Xi}(x) S_{Yj}(y) S_{Zk}(z).$$

There are only 64 non-zero terms in this sum. To specify in which mesh cell the point (x, y, z) is, we define a, b, and c as

$$x \in [x_a, x_{a+1}]; \qquad y \in [y_b, y_{b+1}]; \qquad z \in [z_c, z_{c+1}].$$

Then the 64-term sum can be written

$$\frac{\partial \phi}{\partial x}(x, y, z) = \sum_{i=2a}^{2a+3} \sum_{j=2b}^{2b+3} \sum_{k=2c}^{2c+3} c_{ijk} S'_{Xi}(x) S_{Yj}(y) S_{Zk}(z).$$

From the spline's polynomial definitions (Eq. (1)), the analytical expressions for the spline's derivative can easily be calculated without further approximations.

The polynomial nature of splines is also used to perform accurate integrations. As an example, consider the spatial integration of the potential ϕ inside the mesh,

$$\begin{aligned} I &= \int\int\int \phi(\mathbf{r})\, d^3r \\ &= \int\int\int \sum_{ijk=0}^{N} c_{ijk} S_{Xi}(x) S_{Yj}(y) S_{Zk}(z)\, dx\, dy\, dz \\ &= \sum_{ijk=0}^{N} c_{ijk} \omega_i \omega_j \omega_k \end{aligned}$$

with

$$\omega_i = \int_{x_0}^{x_{N_g}} S_i(x)\,dx; \qquad \omega_j = \int_{y_0}^{y_{N_g}} S_j(y)\,dy; \qquad \omega_k = \int_{k_0}^{z_{N_g}} S_k(z)\,dz. \tag{24}$$

Once again the ωs are determined analytically and involve no further approximations. Furthermore, these ωs depend only on the mesh and thus have to be calculated only once. The calculations of the electrostatic boundary conditions (Eq. (13)) use this integration procedure.

2.4.2. Boundary conditions. This paper will describe only Dirichlet boundary conditons, though other kind of boundary conditions can be handled by BSCM (see [18]). In the case of a localized 3D charge distribution in vacuum, one can obtain an approximation to the potential on the boundaries of the mesh ($\phi_{\alpha\beta\gamma}$) using a multipolar expansion (see, for example, [12]). Up to now, we have used for our physical application an expansion truncated at the quadrupolar term. Depending on specific needs of users, this expansion may include higher-order multipole terms. In any case, the procedure will be a straightforward extension of the one described here. Assuming that the boundaries of the mesh are far enough from the charge distribution, the following formula leads to a good approximation for the potential,

$$\phi(\mathbf{r}) = \frac{Q}{r} + \frac{\mathbf{p}\cdot\mathbf{r}}{r^3} + \sum_{ij=1}^{3} \frac{x_i x_j Q_{ij}}{r^5} + O(1/r^4), \tag{25}$$

with the definitions

$$\begin{aligned} Q &= \int \rho(\mathbf{r})\,d^3r\text{: total charge} \\ \mathbf{p} &= \int \rho(\mathbf{r})\mathbf{r}\,d^3r\text{: dipole moment} \\ Q_{ij} &= \int \rho(\mathbf{r})(3x_i x_j - r^2\delta_{ij})\,d^3r\text{: quadrupolar terms.} \end{aligned} \tag{26}$$

Note that in Eqs. (25) and (26), x_i means x, y, z for $i = 1, 2, 3$, respectively. These quantities are obtained by using a procedure similar to (24) once the interpolation of ρ on the spline basis has been done,

$$\rho(x, y, z) = \sum_{i,j,k=0}^{N} b_{ijk} S_{Xi}(x) S_{Yj}(y) S_{Zk}(z).$$

Then one has to calculate all the multipolar terms such as

$$Q = \sum_{ijk}^{N} \omega_i \omega_j \omega_k b_{ijk}$$

$$p_X = \sum_{ijk}^{N} \omega_i' \omega_j \omega_k b_{ijk}$$

$$Q_{XX} = \sum_{ijk}^{N} (2\omega_i'' \omega_j \omega_z + \omega_i \omega_j'' \omega_k + \omega_i \omega_j \omega_k'') b_{ijk}$$

with the definitions

$$\omega_i = \int_{x_0}^{x_N} S_i(x)\,dx; \qquad \omega_i' = \int_{x_0}^{x_N} x S_i(x)\,dx; \qquad \omega_i'' = \int_{x_0}^{x_N} x^2 S_i(x)\,dx. \tag{27}$$

Once again, expression for these integrals can be found analytically. In practice, it is useful to calculate first the total charge Q and the dipole moment $\mathbf{p}$ and then to evaluate Eq. (25) in the frame of the center of charge $\mathbf{G} = \mathbf{p}/Q$ (for $Q \neq 0$). In this new frame, the dipolar term is obviously zero and ten Eq. (25) can be rewritten

$$\phi(\mathbf{r}') = \frac{Q}{r'} + \sum_{ij=1}^{3} \frac{x_i' x_j' Q_{ij}'}{r'^5}. \tag{28}$$

Note that the quadrupolar term depends on the used frame, $Q_{ij}' \neq Q_{ij}$. However, Q_{ij}' is a simple function of Q, $\mathbf{p}$, and Q_{ij}.

2.5. *Generalizations*

2.5.1. Helmholtz equation. TBSCM is not restricted to Poisson's equation. In fact, if a differential equation involves a differential operator that can be written as a tensorial product of one-dimensional operators invertible using a procedure similar to (21), then the method can be applied. Helmholtz's equation in Cartesian coordinates is the most simple generalization that can be made,

$$[\nabla^2 - \lambda^2]V(\mathbf{r}) = \rho(\mathbf{r}).$$

The corresponding three-dimensional operator $\tilde{A}_{XYZ}$ is

$$\tilde{A}_{XYZ} = \tilde{\nabla}^2_{XYZ} - \lambda^2 \tilde{I}_X \otimes \tilde{I}_Y \otimes \tilde{I}_Z.$$

One can easily verify that

$$\tilde{A}^{-1}_{XYZ} = (\tilde{M}_X \otimes \tilde{M}_Y \otimes \tilde{M}_Z)\tilde{Q}^{-1}_{XYZ}\big(\tilde{M}_X^{-1} \otimes \tilde{M}_Y^{-1} \otimes \tilde{M}_Z^{-1}\big),$$

with

$$\tilde{Q}_{XYZ} = \tilde{\Pi}_X \otimes \tilde{I}_Y \otimes \tilde{I}_Z + \tilde{I}_X \otimes \tilde{\Pi}_Y \otimes \tilde{I}_Z + \tilde{I}_X \otimes \tilde{I}_Y \otimes \tilde{\Pi}_Z - \lambda^2 \tilde{I}_X \otimes \tilde{I}_Y \otimes \tilde{I}_Z,$$

and

$$\big[\tilde{Q}^{-1}_{XYZ}\big]_{\alpha\beta\gamma,\alpha'\beta'\gamma'} = \frac{\delta_{\alpha\alpha'}\delta_{\beta\beta'}\delta_{\gamma\gamma'}}{l_{x\alpha} + l_{y\beta} + l_{z\gamma} - \lambda^2}.$$

Reference [13] describes the tensorial decompositions of other kinds of differential equations.

2.5.2. Quintic splines. Another generalization of TBSCM is the use of higher order polynomial functions such as quintic splines. The main difference in this case is the definitions of these fifth order piecewise polynomial functions. As in the cubic case, a N_g+1 points grid $\{x_a\}$ is defined. On this grid, $3N_g+3$ quintic splines are defined: $\{Q_i(x)\}/i=0\ldots 3N_g+2$. Explicit formulas of these polynomial functions can be deduced from the condensed expression

$$\forall\sigma,\sigma'\in\{0,1,2\}\begin{cases}\frac{\partial^\sigma Q_{3a+\sigma'}(x_b)}{\partial x^\sigma}=\delta_{\sigma,\sigma'}\delta_{a,b}\\ Q_{3a+\sigma}(x)=0 \qquad \forall x\in[x_0,x_{a-1}]\cup\left[x_{a+1},x_{N_g}\right].\end{cases}$$

A function f expanded on this basis is two times differentiable with a continuous second derivative on the grid,

$$f(x)=\sum_{i=0}^{3N_g+2}c_iQ_i(x).$$

In order to find the solution of Eq. (2) as an expansion on a quintic spline basis, one has to determine the $3N_g+3$ unknown variables c_i. In addition to the 2 boundary conditions, $3N_g+1$ collocation points must be defined. In our example we have chosen the following partition: 3 collocation points per grid step (Gauss points) plus one located at the center of the grid (only for even N_g). For a more general approach (n order splines) see [18].

2.6. *BSCM Convergence Order*

Let us define the sum of two Gaussian functions as a model charge distribution,

$$\begin{aligned}g(\mathbf{r})&=(2\pi)^{-\frac{3}{2}}e^{-\frac{r^2}{2}}\\ \rho(\mathbf{r})&=g(\mathbf{r}-\mathbf{r}_1)+g(\mathbf{r}-\mathbf{r}_2)\\ \mathbf{r}_1&=(2.5,2.5,2.5)\\ \mathbf{r}_2&=-\mathbf{r}_1.\end{aligned}\tag{29}$$

A uniform grid is used. The grid's boundaries are

$$\begin{aligned}x_0=y_0=z_0&=-24\\ x_N=y_N=z_N&=24.\end{aligned}$$

Boundary conditions have been fulfilled analytically. Thus, errors are due only to the finite size of the grid step. The analytic expression of the solution of Poisson's equation is

$$V(\mathbf{r})=\frac{\mathrm{Erf}(|\mathbf{r}-\mathbf{r}_1|/\sqrt{2})}{|\mathbf{r}-\mathbf{r}_1|}+\frac{\mathrm{Erf}(|\mathbf{r}-\mathbf{r}_2|/\sqrt{2})}{|\mathbf{r}-\mathbf{r}_2|}.$$

Figure 2 shows the maximum error as a function of the number of collocation points. Sample errors are calculated on a uniform 2D mesh (100×100) in the plane $z=0$. The points $\{\mathbf{r}_i\}$ of this sample are neither collocation nor grid points. Errors are defined by

$$\varepsilon_{\max}(V)=\max_{\{\mathbf{r}_i\}}\{|\phi(\mathbf{r}_i)-V(\mathbf{r}_i)|\}.$$

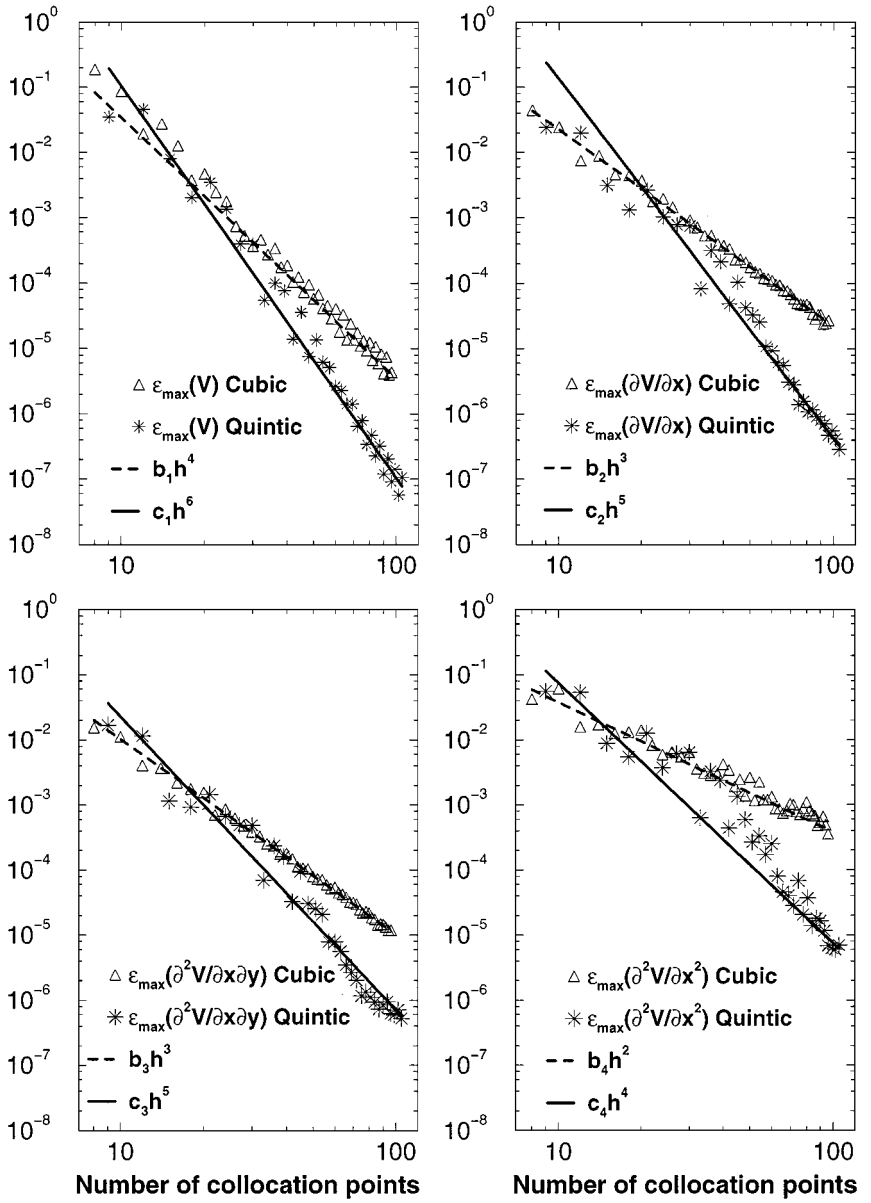

FIG. 2. Maximum error for the approximate potential and its derivatives as a function of the number of collocation points for each space coordinate.

The errors for the derivatives are defined in the same way. In addition to these results, fitted curves have been drawn. Expressions of these polynomial fitting curves are $b_i h^\alpha$ for the cubic case and $c_i h^\beta$ for the quintic case. The b's and c's are the fit parameters, the α and β powers give the orders of the method, while h is the mean-value of the grid step,

$$h = \frac{x_{N_g} - x_0}{N}.$$

The main results of this figure are summarized in Table I. Cubic related results are in

TABLE I
Convergence Order for the Poisson's Equation Solution and Its Derivatives

	V	$\frac{\partial V}{\partial x}$	$\frac{\partial^2 V}{\partial x \partial y}$	$\frac{\partial^2 V}{\partial x^2}$
Cubic splines	$O(h^4)$	$O(h^3)$	$O(h^3)$	$O(h^2)$
Quintic splines	$O(h^6)$	$O(h^5)$	$O(h^5)$	$O(h^4)$

perfect agreement with [11] in which a systematic analysis of the convergence of cubic spline collocation methods in the 2D case is performed. It should be noticed that the second derivative in the quintic case converges as fast as the function V in the cubic case.

As the calculation time is only a function of the number of collocation points, quintic splines seem to be superior and one could wonder why low order splines (cubic splines) can be useful. One obvious reason is that, for low accuracies, the difference between cubic and quintic efficiency is not clear. Another reason can be found by considering the applications of the method. Let us assume that one has to calculate the value of the potential and of the electric field many times everywhere inside the mesh. Actually this would be the case if one had to determine the trajectories of a great number of charged particles moving in the mesh. In such a case, the time needed to evaluate the potential at a given position is a major issue. As we have already seen in Subsection 2.4, such a value can be obtained by a sum over 64 terms in the cubic case. There are $6^3 = 216$ non-zero terms in the corresponding quintic case and this consideration can, in some situations, disqualify high-order splines.

2.7. *Comparison with FFT-Based Method*

The computational cost of TBSCM has been shown to be $12N^4$ (N is the number of collocation points) in Subsection 2.6, while FFT-based methods (see, for example, [10, 5]) have a computational cost proportional to $N^3 \ln(N)$ but rely on the use of uniform grids. Also, N is restricted to be factorizable into a product of small integers, preferentially a power of 2. Because of the multipolar expansion used to find the boundary conditions, the TBSCM can require less floating point operations than FFT-based methods. The following example shall illustrate this point. Let us define the charge density ρ by Eq. (29). The boundary conditions are obtained using Eq. (28). In order to have a good accuracy using this expansion, the following grid boundaries have been chosen,

$$x_0 = y_0 = z_0 = -20$$
$$x_N = y_n = z_N = +20.$$

Let us first define a uniform grid G_1 using these boundaries with 128 collocation points. These boundaries have been chosen in such a way that the errors due to the finite size of the grid step are comparable with those due to the multipolar expansion. A non-uniform grid G_2 is built in order to obtain a finest grid step than G_1 in the charged region and a coarser mesh outside this region. The charge density is almost zero outside a cube defined by

$$x_{1,in} = y_{1,in} = z_{1,in} = -5.2$$
$$x_{2,in} = y_{2,in} = z_{2,in} = +5.2.$$

TABLE II
Compared Accuracies Related to a Uniform (G_1) and a Non-uniform (G_2) Grid

	$\varepsilon(V)$	$\varepsilon\left(\frac{\partial V}{\partial x}\right)$	$\varepsilon\left(\frac{\partial^2 V}{\partial x \partial y}\right)$	$\varepsilon\left(\frac{\partial^2 V}{\partial x^2}\right)$
G_1	$9.0E{-}5$	$1.2E{-}4$	$5.2E{-}5$	$1.0E{-}3$
G_2	$8.7E{-}5$	$3.1E{-}4$	$1.3E{-}4$	$1.6E{-}3$

Inside this cube G_2 is uniform with 26 collocation points. Outside this cube the grid steps grow geometrically from the inner cube to the grid boundaries,

$$\frac{x_{a+1} - x_a}{x_a - x_{a-1}} = r \qquad \forall x_a > x_{2,in}$$
$$\frac{x_a - x_{a-1}}{x_{a+1} - x_a} = r \qquad \forall x_a < x_{1,in}.$$

The first grid step from the outer part of the grid is chosen to be equal to the inner grid step. Thus r depends only on N_2, the number of collocation points in the outer part of the grid. We choose the smallest value for N_2 leading to an accuracy as good as the one obtained with the uniform grid. This accuracy is reached when 10 collocation points are placed to the left and right hand sides of the inner grid part (i.e., $N_2 = 20$). Table II gives accuracies for these two grids.

The prefactors for FFT-based methods depend on specific implementation and algorithms. Moreover, in all cases, those prefactors are larger than the TBSCM ones (i.e., 12). Therefore, we have chosen to remove them from our computational cost estimate. As the order of splines could be different, we assume the same convergence for the finite difference scheme involved in the FFT-based method as the TBSCM one. Obviously, the size required to store the solution is in both cases proportional to N^3. Using these two assumptions the computational costs can be calculated. Table III gives these computational costs.

This example shows clearly (this would have been clearer with prefactors) that the suitability of the grid to the problem is an issue as important as the computational cost. In the case of localized 3D charge distribution in vacuum TBSCM is an efficient method compared to the FFT-based method.

It should be noticed that FFT algorithms can be useful within the TBSCM approach. In [2], the authors use the fact that with cubic splines and uniform grids the eigenvectors and eigenvalues M and Π in Eqs. (19) and (20) are known analytically to apply an FFT algorithm and then reach a $O(N^3 \ln N)$ cost for the TBSCM. An extension of the scope of the FFT-based method to non-uniform grid has been proposed in [8, 9]. Unfortunately

TABLE III
Computational Costs and Storages Related to G_1 and G_2

	Cost (number of operations)	Storage size
G_1 (FFT)	$128^3 \ln 128 = 10.2E6$	$128^3 = 2.1E6$
G_2 (TBSCM)	$46^4 = 4.5E6$	$46^3 = 9.7E4$

this method based on Riemannian geometry makes a direct solving of Poisson's equation difficult and therefore an iterative method is proposed.

The method presented here is direct and straightforward to implement since it involves mainly matrix-vector products. The next section describes the parallelization of the method which greatly benefits from this simple structure. Finally, it is important to note that more complicated methods like multi-grid which scales almost linearly with the number of grid points ($O(N^3)$) are superior to the present approach for large enough grid sizes.

3. PARALLELIZATION OF TBSCM

The TBSCM code seemed to be a good candidate for data parallel programming, and more precisely for high performance Fortran (HPF) programming [6], because it is a regular problem with static control which handles basic data structures (i.e., arrays) with principally linear access to memory. Thus, we decided to port the TBSCM code in high performance Fortran. In the following the sequential program that implements TBSCM is called **Poisson**.

As explained in Section 2, the part of **Poisson** that is the most computationally intensive is the tensor vector product. We first extracted the tensor vector product from the whole code in order to facilitate the test of different parallelization strategies. This appeared to be a good decision since we developed numerous parallel versions of the tensor vector product. In the following, the fragment of code that implements the tensor vector product is called **Tensrus**. We will briefly describe in Subsection 3.2 the main steps of **Tensrus** parallelization. The complete description of the tensrus parallelization is given in [1]. The porting of the whole code in HPF and its performance are presented in Subsection 3.3.

3.1. *Target Machines and Compilers*

During the **Tensrus** parallelization, we used three different MPP Cray machines, a T3D, a T3E-600, and a T3E-750, and mainly the pghpf HPF compiler from Portland Group [17, 4]. Table IV summarizes the characteristics of the target machines. Our experience of parallel programming on the Cray T3D and T3E showed that we generally obtained 10% of their peak performance.

3.2. *Parallelizing the Tensor Vector Product*

We will describe in this section the main steps of **Tensrus** parallelization. **Tensrus** performs exactly $6 \times N^4$ floating-point operations. The Mflops (millions of operations performed in one second) is a good criteria to compare the different parallel versions of **Tensrus**. For a given parallel version of **Tensrus**, Mflops per processor measured with different

TABLE IV
Target Machines Characteristics (1)

System	No. proc.	DECchip	Clock speed MHz	Peak perf. Mflops	MB/PE	Bandwidth Peak (MB/s)
Cray T3D	128	21064	150	150	64	300
Cray T3E-600	128	21164	300	600	128	600
Cray T3E-750	256	21164	375	750	128	600

number of processors evaluates its scalability. Basically, **Tensrus** is composed of three four-dimensional loop nests. The arrays manipulated are two- and three-dimensional. Each loop nest contains three parallel loops and one reduction on the last one. In order to increase the grain parallelism, one has usually to make a parallel loop as computationally intensive as possible and thus to reduce the number of parallel loops. In other words, inside a loop nest the parallel loop is the outer one. This is done simply in HPF by adding a compilation directive **!HPF$ INDEPENDENT** specifying that the loop must be distributed across the processors of the parallel target machine. Then, the data have to be distributed in such a way that the processors own as much as possible of the data needed to perform the computation. In the **Tensrus** program, the three-dimensional arrays manipulated are (**BLOCK, ∗, ∗**) or **∗, ∗, BLOCK**) distributed.[1] The two-dimensional arrays are replicated. In HPF, such array distributions are done by inserting compilation directives in the source code. As an example, the directive **!HPF$ DISTRIBUTE (BLOCK, ∗, ∗) :: fp** means that we want the array **fp** to be distributed according to its first dimension. Thus, each processor will own consecutive lines of **fp**. All the data transfers between processors occurring during the parallel execution of **Tensrus** are due to an array redistribution between the two last loop nests of **Tensrus**.

The performance of this first parallel version of **tensrus** compiled by pghpf 2.1 on a T3D and T3E-600 are the following. The experiments have been performed on 128^3 arrays. On a T3D, the parallel code leads to linear speed-up on 2 to 128 processors with **13** Mflops/PEs and 1.7 GFLOPs on 128 PEs. On a T3E-600, the parallel code leads to linear speed-up on 1 to 64 processors with **28** Mflops/PEs and 2.1 GFLOPs on 128 PEs. These results are already acceptale. But we were not satisfied since the performance of the sequential version was four times better than the performance per processor of the HPF version. Indeed, the sequential version of **tensrus** compiled by the Cray Fortran 90 compiler achieved **74** Mflops on T3D and 136 Mflops on T3E-600.

The reasons for the loss in efficiency have been clearly established. The pghpf complier executes roughly two steps. In the first step pghpf transforms the user's HPF code into a message passing program. Then, during a second step this intermediate message passing program is submitted to the native F90 compiler of the target machine. When the sequential version of **Tensrus** is submitted to the native F90 complier, the complier recognizes that **Tensrus** performs a matrix-vector product. Thus, it replaces the corresponding fragment of code by an optimized routines of the T3x (**SGEMV** for the T3E). When the whole code is submitted to the pghpf compiler, it modifies so much the original code that the native F90 compiler does not recognize any longer the matrix-vector product. Thus, we decided to gather the computations contained in the three loop nests of **Tensrus** (which are "local" computaions) in **PURE** subroutines complied separately with the Cray Fortran 90 native compiler in such a way that it replaces the code fragment that performs a matrix-vector product by a call to the optimized subroutine.

The performance on the T3D is very good. **Tenrus** achieved 8.4 GFLOPS on 128 processors, that is, 66 Mflops per processor. Moreover, the scalability is very good up to 64 processors, where linear speed-up is achieved. The performance on T3E-600 decreases rapidly because there are not enough computations to feed the T3E-600 processors. This is confirmed by the numbers we obtained on larger arrays. As shown in Table V, we obtained

[1] Multi-dimensional distributions are not good data distribution candidates for the **Tensrus** program because they increase the amount of communication and decrease the grain parallelism compared to mono-dimensional data distributions.

TABLE V
PGHPF 2.1: 256^3 Arrays

T3E-600 measures					
No. of proc.	8	16	32	64	128
Mflops	1093	2165	4237	8098	14083
Mflops/PE	136	135	132	126	110

very good scalability with an array size of 256^3 and the performance per processor of the parallel code is very close to that of the sequential version.

It was clear that the last potential source of inefficiency was the necessary redistribution of one of the three-dimensional arrays between the two last loop nests of **Tensrus**. Thus, instead of letting the HPF compiler perform it, we wrote a message passing routine that performs this redistribution. We wrote two versions of the routine. One invoking the message passing interface library [16] and one using the Cray ShMem library [15]. This library is implemented on Cray T3x and SGI Origin systems and provides communications with high bandwidth and low latency. The loss of portability is the main drawback, but with the forthcoming MPI2 standard it will be possible to use the one-sided communications model of ShMem in a portable and efficient way. The "ShMem" version is better than the "MPI" version, scales very well, and achieves very good performance: Table VI summarizes the performances of the final step versions of **Tensrus**.

On 512^3 arrays this version of **Tensrus** (using ShMem) achieves **43** GFLOPs on **128** PEs (i.e., **341** Mflops/PEs) and **85** GFLOPs on **256** PEs (i.e., **332** Mflops/PEs). This represents about **50%** of the T3E-750 peak performance. These results are very satisfactory. Thus, we decided to stop the parallelization of **Tensrus** here.

3.3. *Putting It All Together*

Compared to the parallelization of **Tensrus**, the implementation of the whole code was fast and easy. The parallel code remains very close to the sequential one. Our first task was to add HPF directives for the distribution of all the three-dimensional arrays. Obviously these array distributions correspond to the one specified by **Tensrus**. Then, the 3D and

TABLE VI
HPF-MPI and HPF-ShMem Versions, T3E-750, PGHPF2.3

No. of proc.	2	4	8	16	32	64	128
128^3 arrays							
"MPI" version, Mflops	492	974	1896	3532	6375	8568	11454
"ShMem" version, Mflops	547	1072	2083	4026	7437	13261	17607
"ShMem" version, Mflops/PE	273	268	260	252	232	207	138
256^3 arrays							
"MPI" version, Mflops			2306	4559	8931	16376	29422
"ShMem" version, Mflops			2541	4977	9792	19045	36064
"ShMem" version, Mflops/PE			318	311	306	298	282

TABLE VII
Whole Code, T3E-750, PGHPF2.3

No. of proc.	4	8	16	32	64	128
T3E-750, PGHPF2.3, 128^3 arrays						
TIME (s)	8.08	4.18	2.21	1.17	0.66	
Mflops	824	1576	2912	5344	9088	
Mflops/PE	206	197	182	167	142	
% of Tensrus	82%	82%	79%	75%	69%	
T3E-750, PGHPF2.3, 256^3 arrays						
TIME (s)					7.08	3.82
Mflops					15232	27520
Mflops/PE					232	215
% of Tensrus					82%	77%

4D loop nests (about 20) have been successfully parallelized by adding independent HPF directives. Actually, the only new task consisted of taking into account the fact that, in the parallel version, two kinds of 3D-arrays appear, namely the (**B,** $*$**,** $*$) and the ($*$**,** $*$**, B**). In order to save space, 3D-arrays are used to store different things during the calculations, and therefore we had to optimize their use to avoid redistributions.

The parallel version of **Poisson** runs only with more than 4 (resp. 64) processors on 128^3 (resp. 256^3) arrays because of the memory **Poisson** consumes. We give in Table VII the execution times measured experimentally and the Mflops obtained.[2] Percentages of execution time spent in **Tensrus** are also given. These measurements are also plotted Fig. 3.

3.4. *Parallelization of TBSCM: Concluding Remarks*

The parallel version of **Poisson** achieves nearly linear speed-up until 128 (resp. 32) processors with 256^3 (resp. 128^3) arrays.[3] It achieves *good performance* since nearly 215 Mflops per processor are obtained on a Cray T3E-750 with 256^3 arrays on 128 processors which leads to a total of 27.6 Gflops (27.6 10^9 floating points operations per second).

It is nearly the same code plus only 64 lines of HPF directives and 130 lines of MPI routine calls. The additional lines of code and directives in the parallel version only represent 2% of the total size of **Poisson**. This makes the parallel version of **Poisson** as *readable* as the sequential one. The few parts of code that contain MPI routine calls are enclosed by compilation directives. This allows us to only have one code to maintain instead of two: the sequential and the parallel versions. *Maintainability* of the code is increased. Maintaining only one code that contains both the sequential version and the parallel version is a much harder job with message passing programming. The sequential program has to be transformed in order to take into account the way the work is distributed and

[2] The number of floating point operations performed by **Poisson** is not known exactly, as it is for the **Tensrus** computational kernel. Mflops have been measured with **PAT**, a performance analyzer tool available on T3E. **PAT** has been shown to give accurate measures on various programs (e.g., **Tensrus**). Mflops are given here as landmarks but not as performance measure criteria.

[3] For $n \in [4, 32]$ with 128^3 arrays, the ratio between the execution time on n processors and the execution time on $2 * n$ processors is equal to **1.9**.

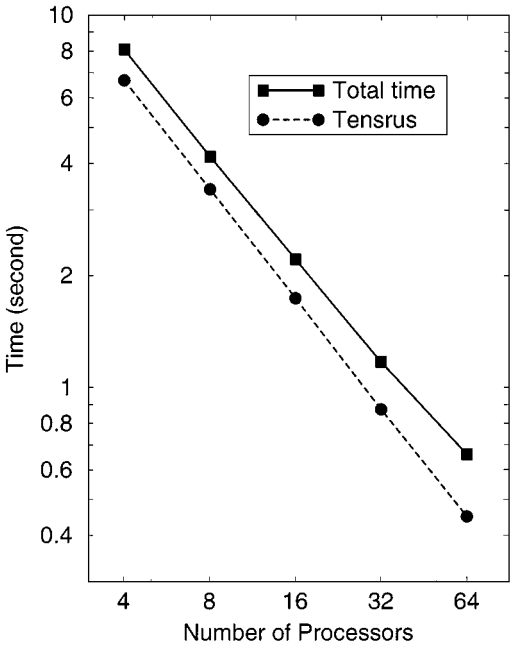

FIG. 3. Whole code time measurements, T3E-750, 128^3 arrays.

the way the data are distributed and communicated between the processes of the parallel execution.

The parallel code achieves *portability* since one efficient version has been written that contains only HPF directives and MPI routines calls.

Correctness of the numerical results, ease of parallel programming, good readability and maintainability of the parallel program, absolute performance, scalability, and portability have been attained.

The parallel implementation of TBSCM that we carried out solves Poisson's equation with 256^3 (resp. 128^3) grids in 3.82 s (resp. 0.66 s) on 128 processors (resp. 64 processors). In practice TBSCM, included in our physical simulation code (Vlasov), is used with a 128^3 grid and 32 processors. Using such a grid was impossible on the sequential workstation (DEC alpha EV56 400 MHz) we used before parallelization, not because of storage but because of CPU time. On this workstation the time required to solve the 128^3 case is 30.5 s. This is 25 times more than using 32 processors on the T3E-750.

4. CONCLUSION

We have given a self-contained description of the TBSCM Poisson solver. The practical calculation of the boundary conditions in the framework of spline basis has been detailed. This solver has been shown to be competitive with FFT-based methods in the case of a localized 3D charge distribution in vacuum because of the non-uniform meshes it can handle. This direct approach has also been shown to be faster than iterative methods. We have emphasized the $O(h^4)$ and $O(h^6)$ convergence of the basis spline collocation method in the cubic and quintic case, respectively. We described finally the parallel implementation of the method and emphasized its high performance and good scalability on a distributed memory T3E Cray machine.

REFERENCES

1. J. Y. Berthou and L. Plagne, Parallel HPF-MPI implementation of the TBSCM Poisson solver, in *Proceedings of HPCN'98, Amsterdam, The Netherland*, Lecture Notes in Computer Science (Springer-Verlag, New York/Berlin, 1998), Vol. 1401.
2. B. Bialecki, G. Fairweather, and K. R. Bennett, Fast direct solvers for piecewise hermite bicubic orthogonal spline collocation equations, *SIAM J. Numer. Anal.* **29**, 156 (1992).
3. C. De Boor, *Practical Guide to Splines* (Springer-Verlag, Berlin/New York, 1978).
4. Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young, Pghpf—An optimizing high performance fortran compiler for distributed memory machines, *Sci. Programming* **6**(1), 29 (1997).
5. E. Braverman, M. Israeli, A. Averbuch, and L. Vozovoi, A fast 3d poisson solver of arbitrary order accuracy, *J. Comput. Phys.* **144**, 109 (1998).
6. *High Performance Fortran Forum: High Performance Fortran Language Specification*, Version 2.0, Technical Report, Center for Research on Parallel Computation, Rice University, Houston, TX, January 1997.
7. C. Guet and L. Plagne, Electron dynamics in metal clusters, in *Theory of Atomic and Molecular Clusters*, edited by J. Jellinek (Springer-Verlag, New York/Berlin, 1999).
8. F. Gygi, Adaptive riemannian metric for plane-wave electronic-structure calculations, *Europhys. Lett.* **19**(7), 617 (1992).
9. F. Gygi, Electronic-structure calculations in adaptive coordinates, *Phys. Rev. B* **48**(16), 11,692 (1993).
10. R. Hockney, A fast direct solution of poisson's equation using fourier analysis, *J. Assoc. Comput. Mach.* **9**, 135 (1965).
11. E. N. Houstis, E. A. Vavalis, and J. R. Rice, Convrgence of $o(h^4)$ cubic spline collocation methods for elliptic partial differential equation, *SIAM J. Numer. Anal.* **25**, 54 (1988).
12. J. D. Jackson, *Classical Electrodynamics* (Wiley, New York, 1975).
13. R. E. Lynch, J. R. Rice, and D. H. Thomas, Direct solution of partial difference equations by tensor product methods, *Numer. Math.* **6**, 185 (1964).
14. L. Plagne and C. Guet, Highly-ionized but weakly excited metal clusters in collisions with multicharged ions, *Phys. Rev. A* **59**(6), 4461 (1999).
15. Cray Research, *Message Passing Toolkit: Release Overview*, ro-5290, 1.1, Technical Report, July.
16. *The MPI Forum*, document for a standard message-passing interface, April 1994.
17. *The Portland Group*, pghpf User's Guide, Version 2.1. Technical Report, May 1996.
18. A. S. Umar, J. Wu, M. R. Strayer, and C. Bottcher, Basis-spline collocation method for the lattice solution of boundary value problems, *J. Comput. Phys.* **93**, 426 (1991).