INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids 2003; **43**:839–864 (DOI: 10.1002/fld.534)

A parallel implementation of the fast multipole method for Maxwell's equations

Pascal Havé*,†

Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie, 4, place Jussieu, 75252 Paris Cedex 05, France

SUMMARY

It is well known that the resolution of Maxwell equations may provide large dense matrices, being thus a computer intensive problem. Even small problems require a huge amount of memory to manipulate matrices during the $O(N^3)$ involved operations.

The fast multipole method enables to *compress* and *approximate* matrices. Coupled with an iterative resolution of the linear system the complexity reduces to $O(N^{\text{iter}} N \log N)$ operations.

In order to use multiprocessors machine and to reduce computation times, we propose here a parallel implementation of the fast multiple method. This article relates our first results, as well as the difficulties encountered. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: fast multipole method; Maxwell; integral formulation; parallel implementation

1. INTRODUCTION

There have been many approaches for the numerical study of Maxwell equations.

Finite Elements [1] or Finite Differences [2] discretize domains between the scattering object and box including the object. They allow multiple frequencies computation with only one simulation and works also with inhomogeneous materials. However, the accuracy may decrease in non-convex domain where the equations have to be integrated for a very long time.

Integral methods [9] compute the electric and magnetic fields on the surface of the object which allows to deduce the value of the entire fields outside the object. They lead to better accuracies than the previous methods, because infinite radiation conditions are well satisfied

*Correspondence to: P. Havé, Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, France.
†E-mail: have@ann.jussieu.fr

Received 3 March 2002
Revised 23 January 2003

and because the frequency domain is used. Moreover, error estimates for Maxwell equations are known for the Raviart–Thomas elements [4] and there opens a way for adaptive solvers.

Solving integral formulations of Maxwell equations may provide large dense matrices. These matrices construction and direct solvers require a lot of memory, and $O(N^3)$ operations, which are already very expensive even for small problems: 6 GB for a small 20 000 unknowns problem! Iterative solutions of the linear systems avoid the storage of the matrices but would be too computer intensive.

An acceleration procedure by Rokhlin [5–7], the fast multipole method, is based on an hierarchical decomposition of the interactions between particles on a surface, using integral equations. The fast multipole method is a way for *compressing* and *approximating* the matrix and for reducing the complexity with an iterative method to an order of $O(N^{\text{iter}} N \log N)$ by a hierarchical decomposition of the interactions between particles. For large problems, this method requires still long computation times and we need a parallel implementation for solving them. Furthermore, clusters of computers become cheap and useful, but an adaptation of the algorithms is needed.

In this paper, first, we will see the mathematical and algorithmic background of the fast multipole method, based on a well-known integral formulation [8].

Then, we will choose some paradigms for our implementation, and explain the problems and how to solve them.

Last, we will provide some first results, a code validation and a performance overview.

2. THE FAST MULTIPOLE METHOD

2.1. Integral formulation

Let us consider the Maxwell's equation in the frequency domain for an electromagnetic monochromatic wave of frequency ω:

$$\begin{aligned}
&\operatorname{rot} E - \imath\omega\mu_0 H = 0 && \text{in } \Omega_e \\
&\operatorname{rot} H + \imath\omega\varepsilon_0 E = 0 && \text{in } \Omega_e \\
&E \wedge n|_\Gamma = -E^{\text{inc}} \wedge n && \text{on } \Gamma \\
&\lim_{|r|\to+\infty} |r|\,|\sqrt{\varepsilon_0}E - \sqrt{\mu_0}H \wedge r| = 0
\end{aligned}$$

where

- E and H are the electric and magnetic vector fields;
- Ω_e is the complement of an open set of boundary Γ assumed perfectly conductive;
- ε_0 and μ_0 are the dielectric parameters of the medium assumed constant;
- n is the normal to the surface Γ;
- $\imath$ is the first square root of -1 in $\mathbf{C}$.

By a theorem of representation [9], we can find the solution to these equations by solving the electric field integral equation or the magnetic field integral equation

EFIE: $\forall y \in \Gamma$,

$$(n \wedge E^{\text{inc}})(y) = \imath\omega\mu_0 \left(\int_\Gamma G(x-y) j(x)\, \mathrm{d}s(x) \right) \wedge n(y) + \frac{\imath}{\omega\varepsilon_0} \left(\int_\Gamma (\nabla_y G(x-y)\, \mathrm{div}_\Gamma\, j(x))\, \mathrm{d}s(x) \right) \wedge n(y)$$

which can be rewritten as, $\forall t \perp n(y)$,

$$-E^{\text{inc}}(y)\, t = \imath\omega\mu_0 \int_\Gamma t\, G(x-y) j(x)\, \mathrm{d}s(x) + \frac{\imath}{\omega\varepsilon_0} \int_\Gamma t\, \nabla_x G(x-y) \mathrm{div}_\Gamma\, j(x)\, \mathrm{d}s(x)$$

MFIE: $\forall y \in \Gamma$,

$$(n \wedge H^{\text{inc}})(y) = \frac{j(y)}{2} + n(y) \wedge \int_\Gamma \nabla_x G(x-y) \wedge j(x)\, \mathrm{d}s(x)$$

where

- j is the electric current (the magnetic field m is zero on a perfectly conductive surface)
- G is the Green kernel defined by

$$G(r) = \frac{\mathrm{e}^{\imath\kappa|r|}}{4\pi|r|} \quad \text{with } \kappa = \omega\sqrt{\varepsilon_0\mu_0} \tag{1}$$

2.2. Discretization

The integral equations are written in weak form, and discretized by the Raviart–Thomas elements.

Let J_i be the Rao–Wilson–Glisson [10] basis functions defined for each edge i at the intersection of two triangles T^+ and T^- by

$$J_i(x) = \begin{cases} \frac{1}{2|T^+|}(x - S^+) & \text{if } x \in T^+ \\ \frac{1}{2|T^-|}(x - S^-) & \text{if } x \in T^- \end{cases}$$

where,

- S^+ and S^- are the vertices opposite to the edge, in T^+ and T^-;
- $|T|$ is the area of a triangle T.

Then the resolution of the EFIE with a Galerkin method based on $\{J_i\}$ leads to solve $Ma = b$ with

$$M_{i,j} = \int_{\Gamma\times\Gamma} G(x-y)(J_i(x) J_j(y) - \frac{1}{\kappa^2} \mathrm{div}_\Gamma\, J_i(x)\, \mathrm{div}_\Gamma\, J_j(y))\, \mathrm{d}\Gamma(x)\, \mathrm{d}\Gamma(y)$$

$$b_i = \frac{\imath}{\omega\mu_0} \int_\Gamma J_i(x)\, E_t^{\text{inc}}(x)\, \mathrm{d}\Gamma(x)$$

Remark 1
M is a symmetric matrix.

The MFIE leads to solve the following linear system, with a Galerkin method:

$$\frac{1}{2}\int_\Gamma J_i(x)J_j(x)\,\mathrm{d}\Gamma(x)+\int_\Gamma J_i(x)\,n(x)\,\mathrm{d}\Gamma(x)\wedge\int_\Gamma \nabla_x G(x-y)\wedge J_j(y)\,\mathrm{d}\Gamma(y)$$
$$=\int_\Gamma J_i(x)\,(n(x)\wedge H^{\mathrm{inc}})\,\mathrm{d}\Gamma(x).$$

However, to achieve a better convergence, we use the combined field integral equation (CFIE), which is a linear combination of EFIE and MFIE defined by

$$\mathrm{CFIE}=\alpha\,\mathrm{EFIE}+(1-\alpha)\frac{\imath}{\kappa}\,\mathrm{MFIE}$$

There exists also more general formulations with a linear combination of EFIE_t, EFIE_n, MFIE_t, MFIE_nterms[‡] [11, 12] for a better performance on several criteria like accuracy, resonance and conditioning number.

These matrices are complex dense matrices and the resolution of the system can be done using an iterative method (e.g. GMRES, QMR, etc.). The main part of the computation is matrix–vector products.

2.3. The FMM algorithm

The FMM is based on the formula

$$\frac{\mathrm{e}^{\imath\kappa|P+M|}}{|P+M|}=\imath\kappa\lim_{l\to+\infty}\int_{S^2}\mathrm{e}^{\imath\kappa\langle s,M\rangle}T_{l,P}(s)\,\mathrm{d}s\quad\text{for } P,M \text{ vectors of } \mathbb{R}^3 \tag{2}$$

where $T_{l,P}$ is the *transfer function* defined for any point $s\in S^2$, the unit sphere, by

$$T_{L,P}(s)=\sum_{m=0}^{L}\frac{(2m+1)\imath^m}{4\pi}\,h_m^{(1)}(\kappa|P|)P_m(\cos(s,P)) \tag{3}$$

with L a truncation parameter, the spherical Hankel function $h_m^{(1)}$, P_m Legendre polynomials, and $\langle\cdot,\cdot\rangle$ the usual scalar product.

Remark 2
Approximations (2) and (3) may increase the computational error, by the fact that:

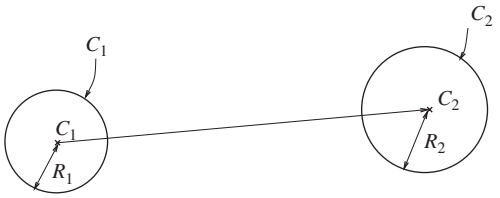
$$\lim_{l\to+\infty}T_{l,P}(s)=+\infty$$

[‡]Respectively, formulations of $E^{\mathrm{inc}}\,t$, $n\wedge E^{\mathrm{inc}}$, $H^{\mathrm{inc}}\,t$ and $n\wedge H^{\mathrm{inc}}$.

We can discretize (2) with a sample of directions s_p of S^2 and associated quadrature weights ω_p

$$\frac{e^{\imath\kappa|P+M|}}{|P+M|} \sim \imath\kappa \sum_p \omega_k e^{\imath\kappa\langle s_p, M\rangle} T_{l,P}(s_p)$$

Let us consider two well separated spheres S_1 and S_2 (of respective centres C_1, C_2 and radius R_1, R_2). For any points x_1 inside S_1 and x_2 inside S_2.



Let us define

$$r_0 = C_2 - C_1, \qquad r = x_1 - C_1 - C_2 - x_2$$

By writing $x_1 - x_2 = r_0 + r$, we obtain

$$\frac{e^{\imath\kappa|x_1-x_2|}}{|x_1-x_2|} \sim \imath\kappa \sum_p \omega_p e^{\imath\kappa\langle s_p, x_1-C_1\rangle} T_{l,r_0}(s_p) e^{-\imath\kappa\langle s_p, x_2-C_2\rangle} \tag{4}$$

2.4. One-level algorithm

Let N be the number of edges on the surface of the perfectly conductive object. The mid-edges are the quadrature points. These points are divided into $\sqrt{N}$ clusters of $\sqrt{N}$ points. Each cluster P_r is bounded by a sphere S_r of centre C_r and radius $R_r = \max_{x_i \in P_r} |x_i - C_r|$.

We define the *well separated* clusters by the negation of

$$P_r \text{ close to } P_s \;\Leftrightarrow\; |C_r - C_s| \leqslant \alpha(R_r + R_s)$$

with $\alpha > 1$ series converge, and $\alpha \geqslant \frac{\sqrt{5}}{2}$ provide a simple control of error (Section 2.6).

Remark 3
Other weaker proximity criteria may be used for a better approximation with larger close-neighbourhood and a larger close-interactions matrix.

We can rewrite the matrix M as the sum of *far interactions* and *close interactions* matrices

$$M_{ij}^{\text{far}} = M_{ij} \quad \text{if } x_i \text{ and } x_j \text{ are not } close \text{ (i.e. } far\text{), else 0}$$
$$M_{ij}^{\text{close}} = M_{ij} \quad \text{if } x_i \text{ and } x_j \text{ are } close \text{ else 0}$$

The FMM method is an efficient way to *compress* the M^{far} matrix with formula (4). To make the product $u = Mv$, we compute $u = u^{\text{far}} + u^{\text{close}}$ with $u^{\text{close}} = M^{\text{close}} v$ and $u^{\text{far}} = M^{\text{far}} v$;

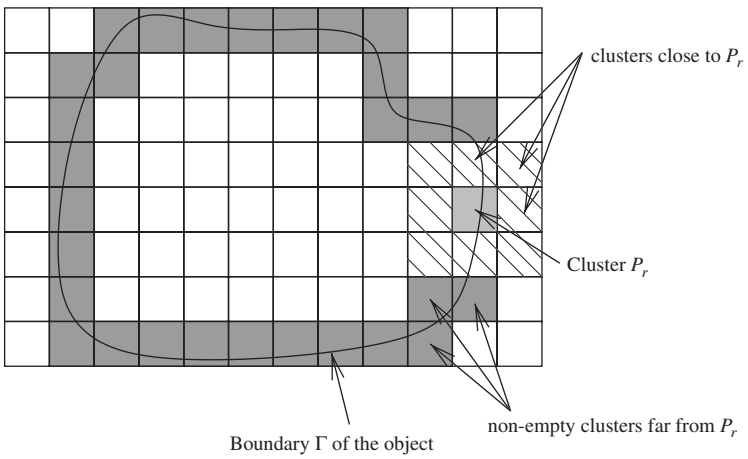


Figure 1. Transfers for one-level FMM.

M^{close} is computed exactly. The FMM compression of M^{far} is a way to compute its product faster but with an approximation and without storing all its coefficients.

We can decompose the computation of $u^{\text{far}} = M^{\text{far}} v$ into 3 steps:

- Computations of radiation functions $F_r(s_p)$ for each cluster P_r

$$F_r(s_p) = \sum_{x_j \in P_r} v_j \mathrm{e}^{\imath\kappa\langle s_p, C_r - x_j\rangle} \tag{5}$$

- Transfer computations between cluster P_r and the others (Figure 1)

$$G_r(s_p) = \sum_{P_t \text{ far from } P_r} F_t(s_p) T_{l, C_r - C_t}(s_p) \tag{6}$$

- Computations of $u_i \in P_r$

$$u_i = \sum_{s_p} \omega_p G_r(s_p) \mathrm{e}^{\imath\kappa\langle s_p, x_i - C_r\rangle} \tag{7}$$

The algorithm and storage complexity are both in an order of $O(N^{3/2})$.

2.5. *Multi-level algorithm*

With the *One-Level* FMM, the number of transfers increases exponentially, when the size of the cluster is reduced (for reducing the size of the *close interactions* matrix). The multi-level algorithm is a hierarchical subdivision of the space giving a hierarchical computational method of transfers.

An efficient way to store the points of the surface of the object, with a hierarchical subdivision, is the oct-tree (Figure 2).

Remark 4

The size of the smallest cubes (leaves of the oct-tree) is $a \sim 1/\kappa$.

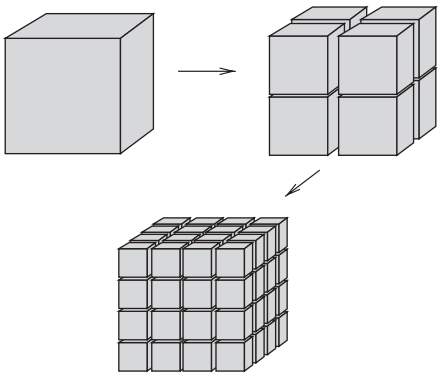


Figure 2. Oct-tree decomposition.

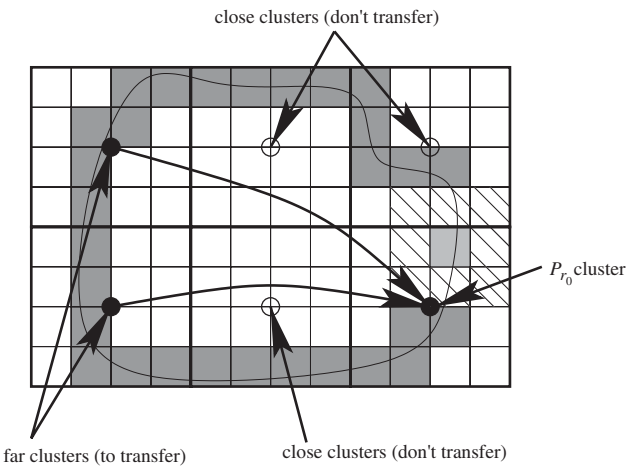


Figure 3. Transfers at level L_0.

For clarity, we shall explain first the multi-level FMM algorithm on the continuous form (2).

Let us consider an oct-tree with 3 levels, named L_0, L_1, L_2. We explain the method for 3 clusters P_{r_0}, P_{r_1} and P_{r_2}, respectively, at the levels L_0, L_1, L_2, such that

$$P_{r_2} \subset P_{r_1} \subset P_{r_0}$$

Like the *One-level* FMM algorithm, there are 3 main steps for the computation. The first is to compute the radiation functions for every cluster P_t at each level.

$$F_t(s) = \sum_{x_j \in P_t} v_j \mathrm{e}^{\imath\kappa\langle s, C_t - x_j\rangle} \tag{8}$$

Then, from the lowest level L_0, we compute the transfers

$$G_{r_0}(s) = \sum_{\substack{P_{t_0} \text{ far from } P_{r_0} \\ P_{r_0} \text{ and } P_{t_0} \text{ at the same level}}} F_{t_0}(s) T_{l, C_{r_0} - C_{t_0}}(s)$$

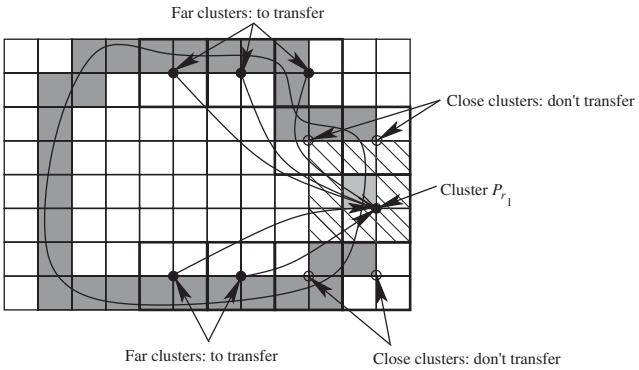


Figure 4. Transfers at level L_1.

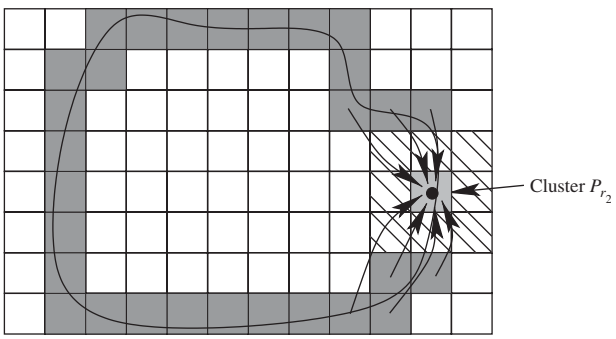


Figure 5. Transfers at level L_2.

Then, at the level L_1 many transfers are already completed, and we need to perform much less transfers (Figure 4).

$$G_{r_1}(s) = \sum_{\substack{P_{t_1} \text{ far from } P_{r_1} \\ P_{t_0} \text{ and } P_{t_0} \text{ are close}}} F_{t_1}(s) T_{l, C_{r_1} - C_{t_1}}(s)$$

where for any P_{t_1} at the level L_1, P_{t_0} denotes its *father*: the only cluster at the level L_0 with $P_{t_1} \subset P_{t_0}$.

We do so up to the higher level, in our example L_2 (Figure 5).

$$G_{r_2}(s) = \sum_{\substack{P_{t_2} \text{ far from } P_{r_2} \\ P_{r_1} \text{ and } P_{t_1} \text{ are close}}} F_{t_2}(s) T_{l, C_{r_2} - C_{t_2}}(s)$$

Then, the last step is to propagate the information from the root (level L_0) to the leaves (level L_2).

$$\tilde{G}_{r_1}(s) = G_{r_1}(s) + \mathrm{e}^{i\kappa\langle s, C_{r_1} - C_{r_0}\rangle} G_{r_0}(s)$$
$$\tilde{G}_{r_2}(s) = G_{r_2}(s) + \mathrm{e}^{i\kappa\langle s, C_{r_2} - C_{r_1}\rangle} \tilde{G}_{r_1}(s)$$

Finally, we obtain $u^{\text{far}} = M^{\text{far}} v$, with the formula

$$u_i^{\text{far}} = \int_{S^2} \mathrm{e}^{\imath\kappa\langle s, x_i - C_{r_2}\rangle} \tilde{G}_{r_2}(s)\, \mathrm{d}s$$

for any $x_i \in P_{r_2}$.

2.6. Multi-level FMM algorithm: discretization

Our discretization of the continuous form (Equation (2)) with quadrature points $\{s_p\}$ on S^2 (Equation (4)) is mainly based on the *exact* integration of the spherical harmonic functions $Y_{l,m}(\theta, \phi)$.

We choose to integrate $Y_{l,m}$ with $\{s_p\} = \{(\theta_j, \phi_i)\}$, for $0 \leqslant i < I$, $0 \leqslant j < J$ with a uniform sample $\phi_i = (2\pi/I)i$ and a Gauss–Legendre sample θ_j, which requires twice less points than with a uniform sample θ_j.

Then, all $Y_{l,m}$, $|m| \leqslant l, 0 \leqslant l \leqslant 2L$ can be integrated exactly with $2L+1$ points for $\{\phi_i\}$ and L points for $\{\theta_j\}$ and the unit sphere is discretized with $(2L+1)L \sim L^2$ points $\{(s_p, \omega_p)\}$, where w_p denotes the associated weight.

Remark 5
Better choices can be made [13], but they may increase the difficulties of an efficient interpolation step.

2.6.1. Harmonic representation. The objective is to represent our discretized formulation (4) applied to the previous discretization as a summation of spherical harmonic functions.

Theorem 1 (*Addition of Spherical Harmonics Functions*)
$\forall l \in \mathbb{Z},\ \forall x, y \in \mathbb{R}^3$

$$P_l(\hat{x} \cdot \hat{y}) = \frac{4\pi}{2l+1} \sum_{m=-l}^{l} \overline{Y_{l,m}}(\hat{x}) Y_{l,m}(\hat{y})$$

where $\hat{x} = x/|x|$.

By an application of the Gegenbauer theorem [14]

$$\mathrm{e}^{\imath x \cdot y} = \sum_{l=0}^{\infty} (2l+1) \imath^l j_l(|x||y|) P_l(\hat{x} \cdot \hat{y}) \tag{9}$$

we can write the transfer function and the exponential function as summation of spherical harmonic functions

$$T_{l,P}(s) = \sum_{l=0}^{L} \imath^l h_l^{(1)}(\kappa|P|) \sum_{m=-l}^{l} \overline{Y_{l,m}}(\hat{P}) Y_{l,m}(\hat{s}) \tag{10}$$

$$\mathrm{e}^{\imath\kappa\langle x, s\rangle} = \sum_{l=0}^{\infty} \imath^l j_l(\kappa|x|) \sum_{m=-l}^{l} \overline{Y_{l,m}}(\hat{x}) Y_{l,m}(\hat{s}) \quad \text{with } s \in S^2 \tag{11}$$

Remark 6
Some experimentations of Chew and Darve [8, 15] show an empirical formula for an acceptable error ε

$$L = \kappa a\sqrt{3} + C(\varepsilon)\log(\kappa a\sqrt{3} + \pi)$$

where a is the size of the cluster.

The same truncation can be applied to (11) with a very low error.

Remark 7

2.6.2. Interpolation. The multi-level FMM algorithm needs discretized spheres S^2 at each level.

First, the computation of $F_{t_i}(s_p)$ can be deduced from $F_{t_{i+1}}(s_p)$

$$F_{t_i}(s_p) = \sum_{\substack{P_r \subset P_{t_i} \\ P_r \in L_{i+1}}} F_r(s_p)\mathrm{e}^{\imath\kappa\langle s_p, C_{t_i} - C_r\rangle}$$

by formula (8) and

$$\mathrm{e}^{\imath\kappa\langle s_p, C_{t_i} - C_r\rangle}\mathrm{e}^{\imath\kappa\langle s_p, C_r - x_i\rangle} = \mathrm{e}^{\imath\kappa\langle s_p, C_{t_i} - x_i\rangle}$$

However, the number discretization points of is not the same at each level, and we know that it is proportional to $(\kappa R_i)^2$ where R_i is the radius of the cluster at the level L_i. At the level L_{i-1}, the number of points is 4 times larger than at the level L_i (because $R_{i-1}^2 = 4R_i^2$), so we need to interpolate.

There are many ways for interpolating such functions.

- Lagrange interpolation for a simple error analysis [16].
- Semi-naive scheme [17], with a forward FFT on ϕ, a dense matrix–vector product on θ, and a backward FFT (exact algorithm, complexity $O(S^{1.5})$, where S denotes the number of points on the unit sphere).
- Scheme due to Alpert–Jakob–Chien [17], as before but the dense-matrix is compressed with a 1D-FMM. Moreover, Yarvin and Rokhlin give an improvement of the 1D-FMM algorithm applicable to this interpolation [18–20] (approximate algorithm, complexity $O(S \log S)$).
- The last one [8] is a sparsification of the one used in the MLFMA variant due to Chew, Lu, Song (complexity $O(S)$).
- A few other algorithms are also available [21–24].

We use the semi-naive scheme, because this step is not so expensive, even for large numbers of unknowns ($\sim 1\,000\,000$).

2.6.3. Anterpolation. In a same manner, the anterpolation allows to reduce the number of points on the sphere at the deepest level from the top to the leaves.

$$\tilde{G}_{r_{i+1}}(s) = G_{r_{i+1}}(s) + \mathrm{e}^{\imath\kappa\langle s, C_{r_{i+1}} - C_{r_i}\rangle}\tilde{G}_{r_i}(s)$$

At the level L_i, there are $4S$ points on the sphere, and at the level L_{i+1}, there are S points. The anterpolation is the symmetric operations of the interpolation. Then, we can use the same methods than the interpolation, with some additional transpositions.

2.7. *Complexity of the algorithm*

For the complexity study, we assume a uniform distribution of Gauss points on the surface.

When we use a one-level FMM algorithm, with N points distributed in $\sqrt{N}$ packets of $\sqrt{N}$ points, the radius of a packet is $O(N^{1/4})$, then the number of points $\{s_p\}$ on S^2 is $O(N^{1/2})$.

The first step of the algorithm provides a $O(\sqrt{N} \times \sqrt{N} \times N^{1/2}) = O(N^{3/2})$ complexity.

The next step, the transfers, needs to transfer all packet couples; the complexity is an order of $O((\sqrt{N})^2 \times N^{1/2}) = O(N^{3/2})$.

The associated storage to the FMM and to the close-interactions are also an order of $O(N^{3/2})$. Indeed, for each packet, the number of close packets is bounded by a constant defined by the proximity criterion, and then for each line of this matrix there are an order of $\sqrt{N}$ non-zero values: there are $O(N \times \sqrt{N}) = O(N^{3/2})$ non-zero values.

The global complexity of the one-level FMM is an order of $O(N^{3/2})$ for the CPU and the storage, but the precomputation of the transfer functions is a way to avoid a additional cost for their recomputation at each iteration.

For the multi-levels algorithm, we need to consider the interpolation, anterpolation and transfer operators. We assume an interpolation operator with a complexity in $O(N \log N)$. Moreover, if we precompute the transfer function ($O(N^{3/2})$ for the higher levels, and very expensive to store), with $O(\log N)$ levels, we obtain a global complexity in $O(N \log^2 N)$.

3. A PARALLEL IMPLEMENTATION

3.1. *Objectives and paradigms*

Our objectives for a parallel implementation of the fast multipole method are:

- High frequency Maxwell equation solver.
  Low frequency scheme and acoustic equation are also planned.
- Distributed memory architecture.
- Adaptive and load balancing capabilities.
- Multiple iterative solvers and preconditioners.

Then, our choices are:

- An object oriented design for a better abstraction of the FMM algorithm.
  The FMM algorithm is independent of the problem: a better re-usability for others equations.
  The language chosen is C++with a heavy usage of *templates*.
- MPI-1, for a message passing paradigm where we used lot of asynchronous communications (and overlapping) in order to hide the latency of the network communications.

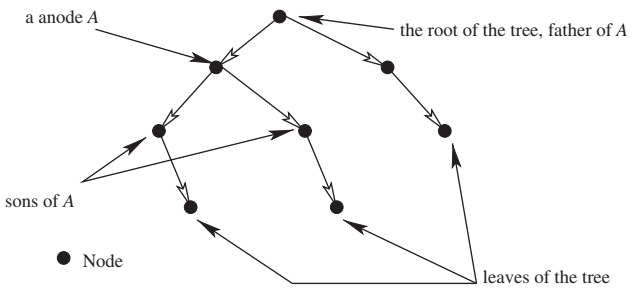


Figure 6.

In the future, we will use MPI-2 for improving the performance on SMP[§] machines, with process spawning on a shared memory machine.

3.2. *Objects distribution*

3.2.1. Definitions. We will define some terms about trees for clarity (Figure 6).

A tree is an oriented graph, where each node may have multiple descendants, and one or zero ascendant.

- *Root*: The *root* is the only node without ascendant (may be also called top of the tree).
- *Father*: The *father* of a node A is its ascendant (the root does not have any father).
- *Son*: A *son* of a node A is one of its descendants.
- *Leaf*: A *leaf* is a node without sons.
- *Cluster*: A *cluster* is an aggregation of triangles.
- *Cell*: The *cell* is the box and the cluster of triangles associated to a node.

3.2.2. The oct-tree. The oct-tree is a fast and easy-to-use manner for ordering the triangles; but we need to find efficiently clusters for many operations like neighbourhood building, transfers. We have chosen to mark each cluster (for each level) with a *unique* label, easy to locate: its location.

A location can be defined with a succession of N-bits numbers, where N is the dimension of the space (each son of the N-tree can be represented with a N-bit number, so each node has 2^N sons).

We define the root of the tree with the location number 1. The location is a mapping of the physical space parameters to a binary number.

However, an oct-tree ($N = 3$) must be bounded by a box. In our case, the box is defined by the FMM algorithm as the smallest cube including all the triangles and with a condition on the size of the smallest cells (of the leaves).

[§]SMP stands for symmetric multi-processors.

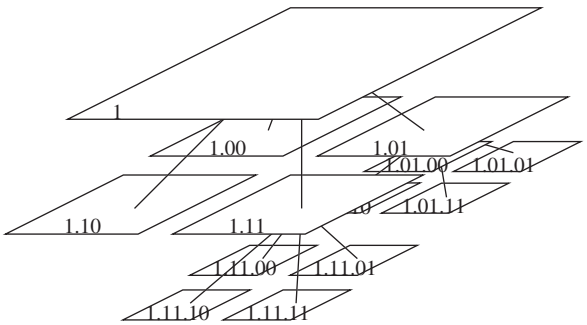


Figure 7. Leaves indexing of a quad-tree ($N = 2$D).

When we need to find the location number of a node, we will use the recursive algorithm:

Algorithm 1 Localization of a point X in a tree

1. Let B_0 be the bounded box of a $l_{\max}$-levels oct-tree
2. Let X be a point of the physical space
3. Let loc_0 be the root location number: 1
4. **for** l from 1 to $l_{\max} - 1$ **do**
5. Compute the relative to B_{l-1} sub-box B_l where X is located
6. We denote s the box-son B_l
7. Each bit of s is the position of the centre B_l relative to the centre of B_{l-1} in one direction (other methods are also allowed)
8. New partial location $\mathrm{loc}_l = \mathrm{loc}_{l-1}.s$
9. **end for**
10. The location of X is $\mathrm{loc}_{l_{\max}-1}$

where '.' denotes the concatenation operator.

Remark 8
This algorithm is also applied for the insertion of the triangles into the oct-tree (Figure 7).

Thus, the number of levels is bounded by the size of the integer used for the representation of the location. A 32 bits number can represent up to 10 3D-levels[¶] of an oct-tree (21 levels for a 64 bits number).

3.2.3. A distributed oct-tree. With a distributed memory computer, we need a distributed oct-tree (a full replicated oct-tree is not efficiently compatible with our future dynamic load balancing).

A way to distribute the tree in order to obtain a fast access to the information with a minimized synchronization, is to replicate (share) a skeleton (the tree without the leaves) and

[¶]Not a real limitation; we have chosen to wrap it into an object which can be increased transparently but it will become more expensive.

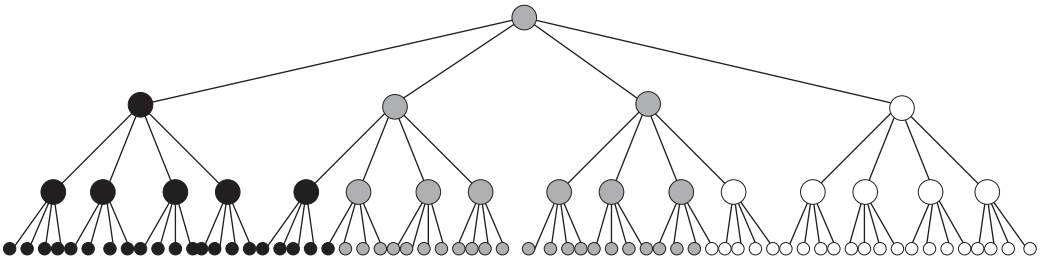


Figure 8. Quad-tree: 3-partitioning skeleton.

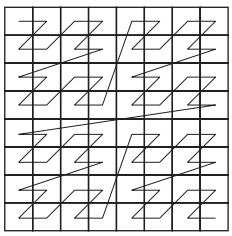


Figure 9. A 3 levels 2D Morton curve.

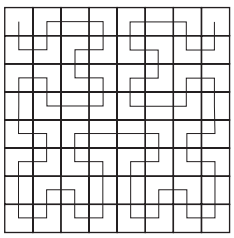


Figure 10. A 3 levels 2D Hilbert curve.

to distribute (not share) the leaves. So, each process knows the existence of data on a leaf, but the value is located on only one memory. This solution must be propagated to all the tree: each node has only one owner (Figure 8).

An algorithm to solve the distribution of data on the tree, is an election algorithm, where the best candidates are the processes with the most number of sons for a given node at the previous level. We do so from the leaves to the top of the tree.

This algorithm is given to minimize the communication to a *repository* node by the other processes, in the case of proximity and hierarchical access. We will see an example in Section 3.4.2.

3.2.4. Surface partitioning. In the case of integral formulations on a closed surface, we need to decompose this surface mesh on several memories, a distributed tree. First, we use an *a priori* partitioning. There are a few well-known methods, often applied to volume partitioning, like the usage of space filling curves (Figures 9 and 10).

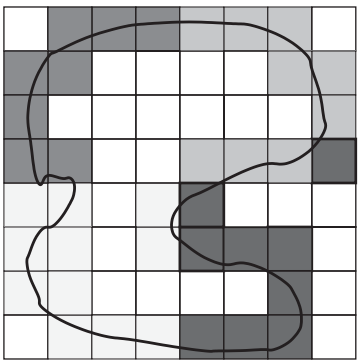


Figure 11. Partition with Morton and Hilbert curves: the same non-simply connected parts; Hilbert curve is not more efficient than Morton curve.

Such methods are mainly based on a linear representation of the space and the algorithm can be summarized like this:

Algorithm 2 Partition of a surface with a space filling curve

1. Let n_T number of triangles, P number of processes
2. Build a space filling curve with l levels
3. **while** Walk along the curve **do**
4. **while** the amount of triangles of the current partition is under n_T/P **do**
5. accumulate triangles
6. **end while**
7. **end while**

Remark 9
The space filling curve can be built on the fly recursively.

Remark 10
In our case, l is given by the FMM algorithm.

We can observe that the Hilbert curve is less efficient when we use it on a surface than on a volume: there are more cases where the partition is not simply connected, like with the Morton curve. However, a good property of this kind of partitioning is to optimize the filling of the tree with a low number of processes, for each tree branch, which reduces close-interaction communications (Figure 11).

Another way may be a partitioning of the surface with its *own* connectivity: to map the surface into a graph. But we need to define criteria for partitioning. What is a good partition for our study ? i.e. one which reduces communications cost.

The communications may be decomposed into two parts:

1. Close interactions;
2. FMM communications: *Broadcast*, *gather*, transfer steps.

To reduce the cost of close interactions, we need to aggregate close clusters at the highest level (i.e. the leaves), like the previous methods.

To reduce the cost of the specific FMM communications, we have to reduce shared parts of the tree between processes, if they deal with far interactions; the *gather* and *broadcast* steps need few communications with our *one owner* convention for nodes distribution. So, a weighted graph partition may be a solution but we need to define what the weights are and how to partition a weighted graph; it is not an easy problem.

3.3. Main algorithms and structures

The FMM algorithm can be decomposed into two parts:

- Close interactions and
- Far interactions stored into a tree.

3.3.1. Close interactions. The close interactions are computed *exactly* and *stored* into a distributed matrix. Four kinds of storage (sorted by speed) can be used.

- Storage into memory (RAM).
- Storage on a external device (like hard drive); the efficiency of this storage depends on the size of the blocks. We have already obtained the same efficiency as full storage in memory with blocks of 30 KB (with SCSI hard drives).
- Half storage: Singular parts are precomputed, left parts are computed *on the fly*.
- No storage, but fully computed *on the fly*, at each iteration.

The close-interactions matrix is a sparse matrix, where only non-zero values are stored. A storage like CSR[‖] matrix is really not efficient, because it makes no use of the cache memory, and has a lot of indirections. An intermediate solution is a compressed sparse block matrix where each coefficient is a full *block matrix* on which we can use fast full matrix product algorithms.[**] Then, we can use a sparse structure for storing only non-zero values, and use efficiently cache memory for very fast full matrix products. Moreover, this way is better for block migration during load-balancing steps, because we do not need to rebuild the entire matrix structure.

The entire product is done with a good overlapping of communications and computations (see Section 3.4 for more details).

Processes	Non-zeros	Local interaction blocks number	Non-local inter. blocks	Timing product (s)	Efficiency (%)
1/CSR matrix	16×10^6	—	—	6.60	—
1	30×10^6	1150×10^3	0	4.08	100
2	15×10^6	577×10^3	2300	2.02	101
4	7.5×10^6	287×10^3	2300	1.00	102
8	3.8×10^6	143×10^3	1700	0.49	104
16	1.9×10^6	71×10^3	2000	0.31	82

[‖]Stands for compressed sparse row matrix also called Morse storage.
[**]We use an optimized BLAS product.

Remark 11
We have obtained these results with 8 dual processors PIII-800 MHz. Up to 8 processes, we use one process on each SMP machine, and for 16 processors, we can observe a decreasing efficiency, maybe, due to an increasing traffic or a worse network management when all processors are working.

Remark 12
This example gives a significantly smaller number of interactions for CSR matrix (here we use a sequential CSR matrix code), but we do not forget the additional memory for rows and columns indices which is on the order of the number of coefficients. However, this example is the worth case for the block storage where each block is very small ($\sim$30 values).

A worst consequence of the choice of the *block partitioning* is to share some degrees of freedom between blocks (local or non-local). Then, an additional step is their synchronization. However, this step is still fast, with the use of overlapping algorithms (there are local and non-local synchronization). We will test a parallel full sparse matrix, for better memory optimization (rather than speed).

Furthermore, the close-interactions storage methods are completely independent of the far-interactions storage methods and allow (in theory) a different partitioning of the close-interactions matrices. This will be used to improve easily the load balancing.

3.3.2. Specific FMM algorithms. The oct-tree is the main support of the data: each leaf supports a part of the discretization (a collection of triangles) and a value (partial value associated to its collection). Moreover, it carries connectivities, like close and far interactions. Far interactions define the transfers between clusters.

At the transfer step, processes exchange values associated to clusters. The mathematical formulation shows them as point-to-point communications.

Even, a sequential implementation uses such a way: doing transfers one by one by reading data to transfer in memory. However, a parallel implementation uses another expensive way: a network. The networks are very slow relative to the memory, and we cannot do that even with buffered communications.

When we analyse globally the transfers step, we can see that one cell sends many transfers to many cells, and at the process level, one cell sends transfers to others cells, but the same data.

Then, we can factorize the communications, so that a cell sends transfers to the others cells by grouping them by process target. However, a received transfer from a cell need to be distributed to local target cells. We use a direct representation for local transfers and a factorized reverse representation for non-local transfers. (the transfers are stored on the target and not on the source).

We need also to store transfers from a received node to the others local nodes; a global point of view is to store transfer on the source node when the target node is local (by symmetry, local transfers are well defined). Each process stores all the skeleton of the tree, even the clusters of the other processes named *virtual* trees where we store local transfers for un-factorization of factorized received transfers (Figures 12 and 13).

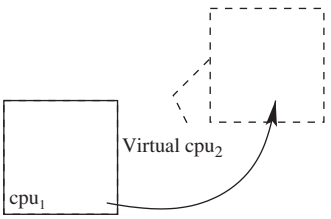


Figure 12. Transfers by cpu_1: factorizing.

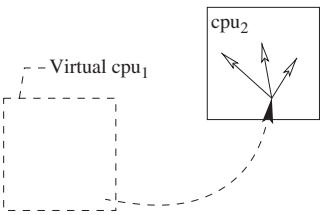


Figure 13. Transfers by cpu_2: un-factorizing.

Then, when we compute transfers, we can list separately local transfers (no communication) and nonlocal transfers (stored on *virtual* trees). This data ordering allows some optimizations seen in Section 3.4.

3.4. Optimization by dynamic overlapping

3.4.1. Task overlapping. The well-known idea is to avoid to wait for the end of communications and using asynchronous communication: we start communications, we do a few *local* computations and we listen to the response of the previous requests. The only problem is how to implement this efficiently.

There are two kinds of overlapping: a static overlapping where the operations are scheduled at compilation time, and a dynamic overlapping where the task loading defines when we need to overlap. A drawback of the first one is the need to foresee when a communication is done and cannot be used while the CPU load is not constant.

A good idea may be to write a multi-threaded program. However, every implementation of MPI are not thread-safe and using it in this context is not so easy.

In fact, we need this for two events.

- Receiving (or waiting for) something and treating the requests.
- Sending something with a possible *pre-processing*.

Furthermore, each communication needs some time for the negotiation of a message with another process: this is the latency. For reducing this, we can bufferize them and reduce the latency of many small messages to the latency of one.

Algorithm 3 A simple way to overlap communications and computations

```
Let InBuffer, OutBuffer be buffers for receiving and sending
while Something to do do
  if there is a ready datum in InBuffer then
    Treat the incoming message
  else if OutBuffer is not full then
    Prepare and send an information with OutBuffer
  else
    Do some local computations
  end if
end while
```

In our implementation, the communication buffers are C++ Objects which negotiate the transport (data conversion and serialization) and the buffering (multiple modes), with the multiple target processes.

Of course, the order of the actions defines their priorities, and can be adapted for each case.

Remark 13
MPI does not provide any warranty about the order of the transmitted messages: we need to have a global point of view of the amount of the communications.

3.4.2. Task lists definition. If we want to apply the previous algorithm to each *critical* function, we need to define what is a *local computation*.

We named a *local computation* a computation without any communication.

We will explain the usage of tasks list by an example.

The chosen example is the reduction of the leaves values to the upper levels: we need to reduce (and interpolate) the value of a node with the values of its sons. This example may be associated to the parallel summation of terms, where each terms are large vectors.

We can define the partial reductions which need to wait for non-local values (and their descendants) as non-local computations. Then, for each computation of this kind, we cannot completely reduce the value and push them in a specific *non-local todo* list. For each computation of this list, the *local* computations do not wait. When a non-local value arrives, the associated treatment is to propagate the value up, until a non-computable node is reached. A non-computable node is a node where there is left at least one value for reducing: the answers of all its sons (local or non-local) are not yet received (Figure 14).

If we can define *local* and *non-local* computations in two different lists (with *local* independent of *non-local* with small granularities), overlapping communications and computations becomes more easy, and efficient.

3.4.3. Limitations. This method is efficient while the time of non-local communications is lower than time of local communications. So, a critical part for the FMM is the transfer step, where the number of transfers increases with the number of level. Some problems may appear when one uses many FMM levels and many processes, because the costs of local computations are really reduced, but the volume of communications increases exponentially. A partial solution is to increase the communication reactivity with more parallel buffers for

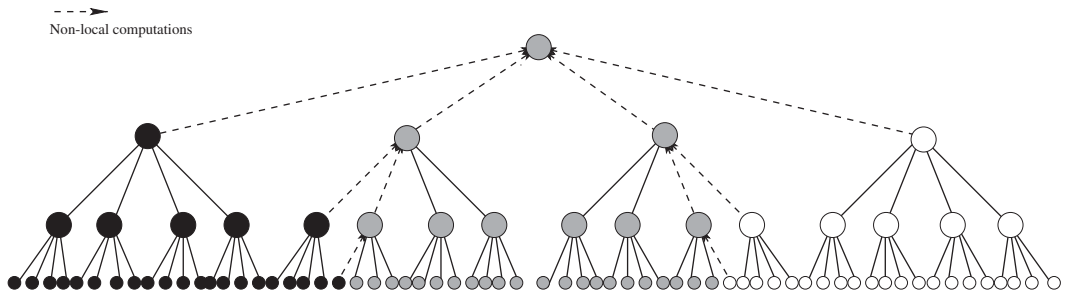


Figure 14. All non-local computations for a 3-partitioning of a quad-tree: reduction case.

sending and receiving data. The number of processes may be bounded by the complexity of the transfer step: if we use too many processes, the efficiency may decrease.

Another weakness may be due to some MPI implementations (and the operating system), the management of the communications, with a high consumption of processor power, or bad reactivity under high cpu loading.

However, the overlapping of communications and computations gives us better load-balancing, with an automatic scheduling of communications during computations.

Moreover, we use this feature only inside functions and not between functions, this will be improved later.

3.5. General design

Here, we explain, in few words, how the different parts are organized

Independent pre-processing and post-processing algorithms:

The pre-processing is decomposed into 4 parts.

- Reading the mesh: triangles, points, triangles orientation;
- Partitioning the mesh according to the best available static load-balancing;
- Building a coherent orientation of the edges: a same arbitrary orientation of each edge seen by all adjacent triangles;
- Building the synchronization structures for shared edges between blocks.

The synchronizing step shares edges from different inner and outer blocks, so as to have an algebrical equivalence to a sequential code. The inner shared edges are due to the strict block partitioning.

Equation specific operators: Each implementation for a given equation needs some specific methods summarized into one specific equation object.

- The FMM operators: interpolation, anterpolation, transfer functions;
- Loading and unloading operators, for loading and unloading values from the current partial solution vector on the FMM-tree leaves, at each iteration of the FMM algorithm;
- The types on the leaves of the FMM tree: an object (often a collection of triangles) and a value (the value associated to the object).

A generic FMM tree: The FMM algorithm is almost the same for every application and equation: *Gather*, *Transfer*, *Broadcast*. For each, we need a specific transfer operator and,

sometimes, specific interpolate and anterpolate operators. This is the goal of our FMM-tree: an abstraction of FMM algorithm, with the management of the parallelism, and calls to specific external operators (via *templates*). The *factorized transfers* feature (Section 3.3.2) is use by default.

Many new equations may be solved with only a modification of these specific operators without changing the global FMM algorithm. However, the transfer step may be different when we use a different transfer communication scheme (like a non-factorizable transfers step).

Generic distributed sparse block matrices and solvers: The iterative scheme involves solvers and some linear algebra. They are not specific to the fast multipole method. We implemented various kinds of block storage: *in core*, *out of core*, *recomputed*, then an abstraction of the parallel sparse block matrix allows a generic usage with a good efficiency and for a small effort.

At the present time, we are using two iterative solvers: GMRES and Deflated GMRES. They are independent of the parallel linear algebra; more solvers are planned.

The use of good preconditioners (SPAI [25, 26]) may increase the efficiency of the implementation on cases with bad condition numbers.

4. NUMERICAL RESULTS

Our code is not yet fully optimized and tuned for the best parameters (buffer size, scheduling algorithm, etc.), but some results are already satisfactory.

4.1. Unit radius spheres: code validation

This example is a test case for the validation; a comparison is done with the exact solution of Maxwell's equation for a perfectly conductive unit radius sphere.

For this test, we have used $\mu_0 = 1.25751 \times 10^{-6}$, $\varepsilon_0 = 8.84806 \times 10^{-12}$ giving $c = 299792548.2$ m/s.

The sphere at $\lambda = 0.3$ *with* 50 000 *triangles*: The mesh has 50 000 triangles and corresponds to 75 000 edges. At a frequency $f = 10^9$ Hz corresponds to a wave number $\kappa = 21$ and to a wavelength $\lambda = 0.3$, there are approximately 10–12 points by wavelength. We have used 7 FMM Levels. All this gives 11 400 000 non-zero coefficients (96 000 block interactions) and 420 000 transfers (Figure 15).

The same problem with 200K triangles and $\lambda = 0.15$: The mesh has 200 000 triangles (i.e. 300 000 edges). A frequency $f = 2 \times 10^9$ Hz giving a wave number $\kappa = 42$ and a wavelength $\lambda = 0.15$. Again this corresponds to 10–12 points by wavelength. We have used 8 FMM levels, giving 80 000 000 non-zero coefficients (106 000 block interactions) and 1 600 000 transfers.

Performance overview: These tests are done on a cluster of 8 Dual PIII-800 MHz SMP with 1 GB of memory for each. The cost of close interactions matrix products are not included in these results, because the *in-core* mode does not allow to use a low number of processors and the *out-of-core* mode is dramatically bad when 2 processes use the same hard drive for storing the matrices. Furthermore, we do not count the synchronization step, because it may decrease the results without load balancing features (not well equilibrated loads, especially when the number of processors is large and not a power of 2) (Figure 16).

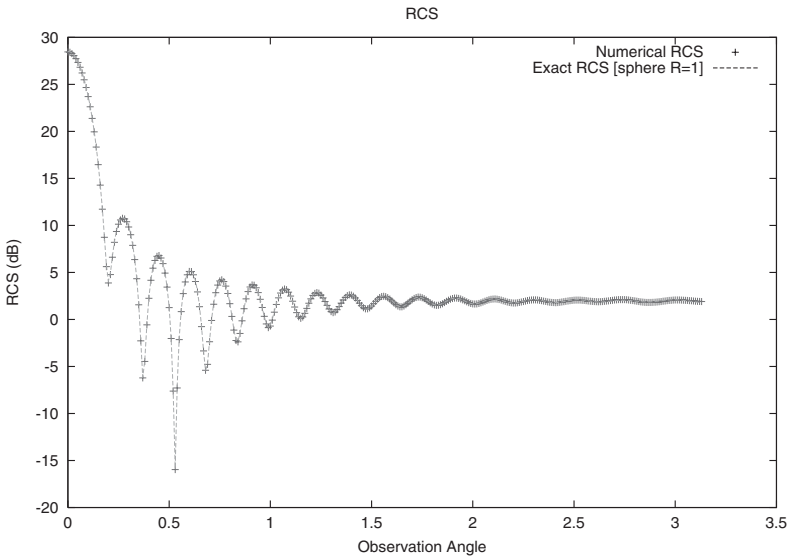


Figure 15. 1 GHz, 50 K triangles, unit radius sphere test case.

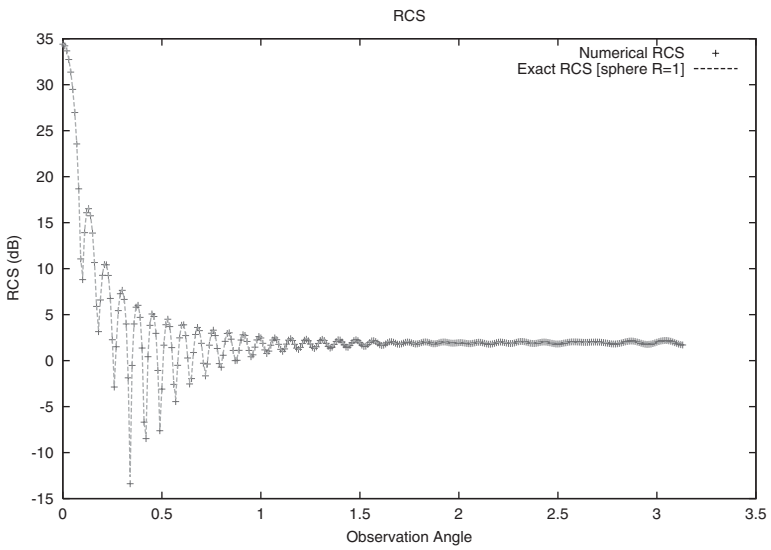


Figure 16. The unit radius sphere test case at 2 GHz and 200 K Triangles.

Remark 14
The efficiency may be defined as the complement of the relative loss of computational power due to increasing the number of processors (Figures 17 and 18).

600 *K unknowns case*: The mesh has 400 000 triangles (i.e. 600 000 edges). A frequency $f = 3 \times 10^9$Hz giving a wave number $\kappa = 63$ and a wave length $\lambda = 0.1$. Again this corresponds

Processors	1 Iteration Time	Transfer Eff.	w/out Transfer Eff.	Global Efficiency
1	460s	-	-	-
2	270s	63%	98%	85.3%
4	143s	54%	98%	80.6%
8	80s	41%	97%	71.6%
12	80s	21%	86%	47.4%
16	63s	20%	82%	45.6%

Figure 17. 300 K–2 GHz unknowns case (*uses the previous* 200 K *Triangles on a unit radius sphere*).

Processors	1 Iteration Time	Transfer Eff .	w/out Transfer Eff .	Global Efficiency
2	141s	-	-	-
4	73s	83%	99%	96.4%
8	40s	63%	97%	89.1%
12	33s	40%	85%	71.9%
16	29s	25%	80%	59.0%

Figure 18. 300 K–1 GHz unknowns case.

to 10–12 points by wavelength.

Processors	1 Iteration Time	Transfer Eff.	w/out Transfer Eff.	Global Efficiency
2	564s	—	—	—
4	302s	85%	99%	93.5%
8	164s	71%	98%	96.2%
12	145s	45%	88%	65.0%
16	115s	41%	85%	61.2%

Comments. The computations are divided into 2 parts: fully scalable, and badly scalable. Many parts of our code are really scalable, but the critical part is the transfer step. This step is mainly communication oriented, and the overlapping of communications with local computations is more complicated. Even, using factorized transfers allows to reduce communications with more local computations and increase the overlapping, but when the number of processors is high, there are more transfers, and the efficiency decreases. We are studying for an optimization of the transfers step by an adaptive loop size (i.e. growing local computations); some *good* results are already available (up to $\sim 20\%$ better: do not stress the network card). However, these algorithms are not always stable, because the functional is really chaotic (depends on time).

We can see worse performances when the number of processors is above 8 (more than 1 CPU used on each SMP machine). This may be due to the SMP architecture with a slow network (one Ethernet 100 MB/s card for 2 processors), and the cache memory invalidation (2 separated memory management units).

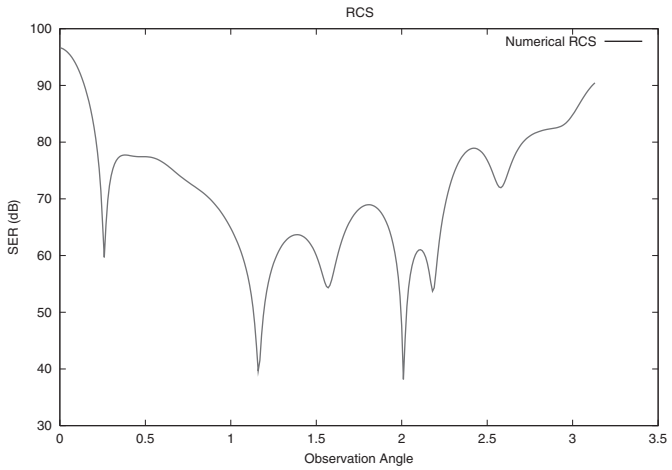


Figure 19. 6λ Airplane: radar cross-section.

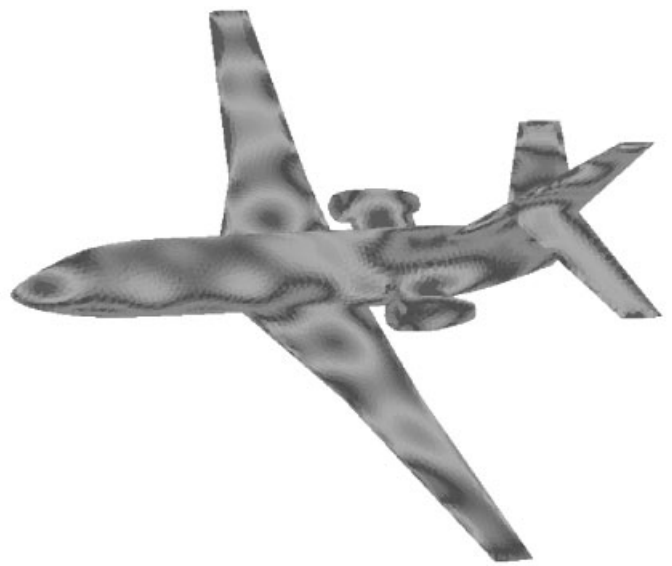


Figure 20. 6λ Airplane: intensity of the electric field.

4.2. An airplane at 6λ

We have used a non-homogeneous mesh with 30 000 edges giving 10–18 points by wavelength and 7 FMM Levels (Figures 19 and 20).

5. CONCLUSION

Our implementation is not yet fully featured for solving very large problems (preconditioners, etc.) [27, 28]; indeed, the main limitation is the number of levels which provides very large transfer function to store in memory. Some optimizations with a computation *on the fly* or a storage on a external memory are possible, but an optimal reloading of partial transfer function in main memory is not so easy due to the overlapping of computations and communications (a communication may be in late, and so need to reload twice the same transfer function). Another solution may be to *compress* transfer functions [27]. For a better performance evaluation, we need to use more realistic case than the unit sphere and a good (static and dynamic) load balancing will be necessary to avoid spending time waiting for overloaded processes.

Nevertheless, we had implemented a number of features for the parallelization of the Fast Multipole Method and we have obtained very encouraging and efficient performances.

ACKNOWLEDGEMENTS

I thank Eric Darve for his support on the Fast Multipole Methods, which is the foundation of my FMM understanding. This work is in continuation of his PhD thesis [8].

I thank also Olivier Pironneau for PhD guidance.

REFERENCES

1. El Dabaghi F, Morgan K, Parrott K, Periaux J (eds). *Approximations and Numerical Methods for the Solution of Maxwell's Equations*. The Clarendon Press Oxford University Press: New York, 1998.
2. Yee KS. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Transactions on Antennas and Propagation* 1966; **14**:302–307.
3. Nédélec JC. *Approximation of integral equations by finite elements*, 1990.
4. Buffa MCA, Schwab C. Boundary element methods for Maxwell's equations on non-smooth domains. *Report* 2001–01, 2001.
5. Engheta N, Murphy W, Rokhlin V, Vassiliou MS. The fast multipole method (fmm) for electromagnetic scattering problems. *IEEE Transactions on Antennas and Propagation* 1992; **40**:634–641.
6. Coifman R, Rokhlin V, Wandzura S. The fast multipole method for the wave equation: A pedestrian prescription. *IEEE Antennas and Propagation Magazine* 1993; **35**:7–12.
7. Rokhlin V. Rapid solution of integral equations of scattering theory in two dimensions. *Journal of Computational Physics* 1990; **86**:414–439.
8. Darve E. Méthodes multipôles rapides: résolution des équations de Maxwell par formulations intégrales. *Ph.D. Thesis*, Université Pierre et Marie Curie, 1999.
9. Nédélec JC. *Cours de DEA de l'école polytechnique et de l'université Pierre erMarie Curie, Paris* 1999; **6**.
10. Rao SM, Wilton DR, Glisson AW. Electromagnetic scattering by surfaces of arbitrary shape. *IEEE Transactions on Antennas and Propagations* 1982; **AP-30**:409–418.
11. Sheng X-Q, Jin J-M, Song J, Lu C-C, Chew W. Multilevel fast mulitpole algorithm for electromagnetic scattering by large complex objects. *IEEE Transactions on Antennas and Propagations* 1997; **45**:1488–1493.
12. Sheng X-Q, Jin J-M, Song J, Lu C-C, Chew W. Solution of combined-field integral equation using multilevel fast multipole algorithm for scattering by homogeneous bodies. *IEEE Transactions on Antennas and Propagations* 1998; **46**:1718–1726.
13. McLaren AD. Optimal numerical integration on a sphere. *Mathematics of Computation* 1963; **17**:361–383.
14. Abramovitz M, Stegun I. *Handbook of Mathematical Functions*. Applied Mathematical Series. National Bureau of Standards: Washington, DC, 1964.
15. Song J, Lu C-C, Chew W. Multilevel fast multipole algorithm for electromagnetic scattering by large complex objects. *IEEE Transactions on Antennas and Propagations* 1997; **45**:1488–1493.
16. Koc S, Song J, Chew W. Error analysis for the numerical evaluation of the diagonal forms of the scalar spherical addition theorem. *Research Report CCEM*-32-97, University of Illinois, Urbana, IL 61801-2991, October 8 1997.
17. Alpert B, Jakob-Chien R. A fast spherical filter with uniform resolution. *Journal of Computational Physics* 1997; **136**:580.

18. Yarvin N, Rokhlin V. An improved fast multipole algorithm for potential fields on the line. *SIAM Journal on Numerical Analysis* 1999; **36**:629–666.
19. Yarvin N, Rokhlin V. A generalized one-dimensional fast multipole method with application to filtering of spherical harmonics. *Journal of Computational Physics* 1998; **147**:594–609.
20. Dutt A, Gu M, Rokhlin V. Fast algorithms for polynomial interpolation, integration and differentiation. *SIAM Journal of Numerical Analysis* 1996; **33**:1689–1711.
21. Bucci O, Gennarelli C, Riccio R, Savarese C. Electromagnetic fields interpolation from nonuniform samples over spherical and cylindrical surfaces. *IEEE Proceedings on Microwave Antennas and Propagation* 1994; **141**:77–84.
22. Bucci O, Gennarelli C, Savarese C. Optimal interpolation of radiated fields over a sphere. *IEEE Transactions on Antennas and Propagation* 1991; **AP-39**:1633–1643.
23. Knab JJ. Interpolation of band-limited functions using the approximate prolate series. *IEEE Transactions of the Information Theory* 1979; **25**(6):717–720.
24. Knab JJ. The sampling window. *IEEE Transactions of the Information Theory* 1983; **29**(1):157–159.
25. Grote MJ, Huckle T. Parallel preconditioning with sparse approximate inverses. *SIAM Journal of Science and Computation* 1997; **18**:838–853.
26. Alléon G, Benzi M, Giraud L. Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics. *Technical Report* TR/PA/97/05, CERFACS, Toulouse, France, April, 1997.
27. Velamparambil S, Chew WC. A fast polynomial representation for the translation operators of MLFMA, 2000.
28. Velamparambil S, Chew WC. Parallelization of MLFMA on distributed memory computers. In *Proceedings of the International Conference on Electromagnetics in Advanced Applications (ICEAA01)*, Roberto D Graglia (ed.). pp. 141–144. 10–14 September 2001 (Expanded version under the review of "Parallel Computing").

