Available online at www.sciencedirect.com

ScienceDirect

Journal of Computational Physics 227 (2008) 1836–1862

JOURNAL OF COMPUTATIONAL PHYSICS

www.elsevier.com/locate/jcp

# High performance BLAS formulation of the multipole-to-local operator in the fast multipole method

O. Coulaud [a], P. Fortin [a,b,*], J. Roman [a]

[a] ScAlApplix Project, INRIA Futurs, LaBRI and IMB, Université Bordeaux 1, 351 cours de la Libération, 33405 Talence Cedex, France
[b] Laboratoire d'Astrophysique de Marseille, 2 place Leverrier, 13248 Marseille Cedex 4, France

Received 20 December 2006; received in revised form 15 September 2007; accepted 29 September 2007
Available online 12 October 2007

**Abstract**

The multipole-to-local (M2L) operator is the most time-consuming part of the far field computation in the fast multipole method for Laplace equation. Its natural expression, though commonly used, does not respect a sharp error bound: we here first prove the correctness of a second expression. We then propose a matrix formulation implemented with basic linear algebra subprograms (BLAS) routines in order to speed up its computation for these two expressions. We also introduce special data storages in memory to gain greater computational efficiency. This BLAS scheme is finally compared, for uniform distributions, to other M2L improvements such as block FFT, FFT with polynomial scaling, rotations and plane wave expansions. When considering runtime, extra memory storage, numerical stability and common precisions for Laplace equation, the BLAS version appears as the best one.
© 2007 Elsevier Inc. All rights reserved.

*MSC:* 65Y20; 70F10

*Keywords:* Fast multipole methods; Laplace equation; BLAS routines; Error bound; Fast Fourier transform; Rotations; Plane waves; Uniform distribution

## 1. Introduction

The $N$-body problem in numerical simulations describes the computation of all pairwise interactions among $N$ bodies. The fast multipole method (FMM) developed by Greengard and Rokhlin [1,2] solves this $N$-body problem for any given precision with $\mathcal{O}(N)$ runtime complexity against $\mathcal{O}(N^2)$ for the direct computation. The potential field is indeed decomposed in a near field part, directly computed, and a far field part approximated thanks to multipole and local expansions. First introduced for gravitational potentials in astrodynamics or

* Corresponding author. Address: Laboratoire d'Astrophysique de Marseille, 2 place Leverrier, 13248 Marseille Cedex 4, France.
*E-mail addresses:* olivier.coulaud@inria.fr (O. Coulaud), pierre.fortin@lip6.fr (P. Fortin), roman@labri.fr (J. Roman).

0021-9991/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2007.09.027

electrostatic (coulombic) potentials in molecular dynamics [1,3], it has then been extended with different mathematical bases to electromagnetism [4], VLSI capacitance [5], radiosity [6], object modeling [7] and many more. In this paper we will focus on the *Laplace equation* and we will therefore only consider *gravitational* and *electrostatic potentials*.

In the FMM computation for Laplace equation the multipole-to-local operator (*M2L*), which converts a multipole expansion into a local expansion, represents most of the runtime of the far field computation. The operation count of this operator is $\mathcal{O}(P^4)$ in the $3D$ FMM, where $P$ is the maximum degree in the expansions. In order to fasten its computation, some schemes have been introduced that reduce the $\mathcal{O}(P^4)$ operation count to $\mathcal{O}(P^3)$, $\mathcal{O}(P^2 \log P)$ or $\mathcal{O}(P^2)$: block fast Fourier transform (FFT) [8], FFT with polynomial scaling [9], rotations [10] and more recently plane wave expansions [11,12]. When considering operation count and numerical stability, the plane wave improvement appears to be the best one, but to our knowledge no other work has been published about this plane wave scheme with Laplace equation[1] since the article [12]. The FFT and rotation schemes are thus still used in this context: see [14,15].

However, on modern architectures the computation speed of the processor is much higher than the speed of memory access. This difference leads to important waiting times in order to access data in memory: the processor is then unused, which dramatically affects the CPU times of these simulations. This is why we propose here a different approach: we will use highly efficient implementation techniques such as BLAS (Basic Linear Algebra Subprograms) routines [16] to improve the runtime of the FMM. Thanks to optimal use of the different layers of the hierarchical memory of the computer, so that the pipelines of the floating point units are filled at best, they indeed offer substantial runtime speedup on superscalar architectures. This speedup only affects the constant in the $\mathcal{O}(P^4)$ notation and we keep the $\mathcal{O}(P^4)$ operation count. But in molecular dynamic simulations for example, the required precisions for electrostatic potentials range usually between $10^{-5}$ and $10^{-7}$, so $P$ ranges from 3 to 15 (see [3] or [17]); and for gravitational potentials that arise in astrophysics the required precisions and the $P$ values are even lower [18]. All these values of $P$ are quite low in terms of operation count, and the speedup obtained with the BLAS routines can thus exceed the one obtained with a scheme that offers a lower operation count.

We also distinguish two different expressions of the *M2L* operator. Indeed, the error bound of the 3D FMM has been historically [1,2] presented for the evaluation of potential with either finite multipole expansions or finite local expansions. But, as mentioned by several authors [19,20], we have also to consider, when implementing the FMM, that the *M2L* operator acts on finite multipole expansions, which means that both multipole and local expansions are finite. When denoting $P$ the maximum degree of the expansions, and $n$ (respectively $j$) the degrees of the multipole (respectively local) expansion terms, two different kinds of *M2L* expressions can then be used. In the first one, both $n$ and $j$ fully range between 0 and $P$, whereas in the second *M2L* expression we use only terms with $n + j \leqslant P$. While the first one is natural and commonly used ([19–21]), no sharp error bound has yet been found, to our knowledge, for the corresponding summations. We will prove that the second one, though generally less efficient, respects such sharp error bound. For these two expressions, we will present the principles and the implementation features for three improvements of the *M2L* operator, namely block FFT, FFT with polynomial scaling and rotations, as well as for our BLAS approach. We will then propose a detailed comparison of the memory requirements, numerical precisions and runtimes, for uniform distributions sequentially computed, between these schemes depending on the *M2L* expression used.

Thanks to the FMMPART3D code (version 1.0) distributed by the MadMax Optics company,[2] we will also compare our BLAS version with the enhancement based on plane wave expansions. To our knowledge, this is the first comparison between this scheme with plane waves and another improvement of the *M2L* operator when considering the FMM for Laplace equation, since only comparisons with direct computation are given in [11,12].

The rest of this article is organised as follows: in the next section we will introduce the formulae used in our implementation of the FMM. We will also discuss one error bound issue and present the two different versions of the *M2L* operator. We will expose and discuss the FFT enhancements in Section 2.3, the use of rotations in

[1] The plane wave scheme has however been widely used for Maxwell and Helmholtz equations (see [13] for example) since the $P$ values are much higher in these cases.

[2] http://www.madmaxoptics.com/technology/products/FMMPART3D.html.

Section 2.4, the plane wave scheme in Section 2.5 and the introduction of BLAS routines in Section 3. The FFT, rotation and BLAS improvements have been implemented in a code named FMB for fast multipole with BLAS (or fast multipole in Bordeaux) thus enabling precise comparisons among them as presented in Section 4. For almost all these sections, more details and explanations can be found in the research report [22]. The current article has to be seen as a summary of this research report, except for the final comparison in Section 4.5 between the scheme with plane waves and our BLAS approach.

## 2. Multipole to local operator: presentation and current improvements

We present here the *M2L* operator, its mathematical formulae, the two possible expressions, their consequence on the error bound and three current improvements of its operation count that we have implemented: the block FFT, the FFT with polynomial scaling and the rotations. The plane wave scheme is also briefly presented.

### 2.1. FMM presentation

We focus in this article on uniform distributions sequentially computed and refer the reader to the articles of Greengard and Rokhlin for a presentation of the FMM and especially the notions of quadtree and octree, near and far fields, multipole and local expansions, upward and downward passes, nearest neighbors, *interaction list*, and *well-separateness*. We denote *ws* the well-separateness criterion (see [20]) and use $ws = 1$ here, which means that two cells of the octree are well-separated provided they do not share a boundary point. The interaction list of a cell $c$ is then defined as the set of all children of the nearest neighbors of the parent cell of $c$ which are not themselves nearest neighbors of $c$; there is 189 members in each interaction list.

We denote by *M2M* the operator that translates a multipole expansion. *M2L* denotes the conversion of a multipole expansion into a local expansion, and *L2L* the translation of a local expansion. We consider the root of the octree (or *computational box*) to be at level 0. The *height* of the octree is defined as its maximum level. Moreover we use Morton ordering for the indexing of each cell: this enables a fast access to all cells in the octree thanks to bit operations (see [3]).

We introduce now briefly the formulae used in our implementation of the FMM focusing on the *M2L* operator: more details and the complete formulae set can be found in [22]. We have chosen the formulae of Epton and Dembart [23] for the clarity of the underlying theorems and their proofs, and also because they enable elementary formulation of the *M2L* operator, which represents the most time-consuming part of the far field computation, while obeying the symmetry property among the multipole and local expansion terms of opposite order (see Lemma 1 at the end of the section).

In 3D space, $\theta$ denotes the co-latitudinal coordinate and $\phi$ the longitudinal coordinate. With $\epsilon_l = (-1)^l$ if $l \geqslant 0$, and $\epsilon_l = 1$ otherwise, and $P_n^l$ denoting the associated Legendre function, our unnormalized spherical harmonics $Y_n^l$ of *degree* $n$ and *order* $l$, with $n \geqslant 0$ and $-n \leqslant l \leqslant n$, are defined by

$$Y_n^l(\theta,\phi) = \epsilon_l \left(\frac{(n-l)!}{(n+l)!}\right)^{\frac{1}{2}} P_n^l(\cos\theta)\mathrm{e}^{\mathrm{i}l\phi}. \tag{1}$$

We emphasize that our spherical harmonics are considered as identically null for $n < 0$ or $|l| > n$. Like Epton and Dembart [23], we also respectively define the *Outer* and *Inner* functions by

$$O_n^l(r,\theta,\phi) = \frac{(-1)^n i^{|l|}}{A_n^l} Y_n^l(\theta,\phi)\frac{1}{r^{n+1}} \quad \forall (n,l) \in \mathbb{N}\times\mathbb{Z} \quad \text{with } |l| \leqslant n \tag{2}$$

and

$$I_n^l(r,\theta,\phi) = i^{-|l|} A_n^l Y_n^l(\theta,\phi) r^n \quad \forall (n,l) \in \mathbb{N}\times\mathbb{Z} \quad \text{with } |l| \leqslant n, \tag{3}$$

where $A_n^l = \frac{(-1)^n}{\sqrt{(n-l)!(n+l)!}}$.

Letting $\mathbf{X}$ and $\mathbf{X}'$ be two position vectors in 3D space, we now give the main theorems that the FMM requires: their proof can be found in [23].

**Theorem 1** (Classical translation theorem). *Under the assumption* $\|\mathbf{X}\| > \|\mathbf{X}'\|$, *we have*

$$\frac{1}{\|\mathbf{X}-\mathbf{X}'\|} = \sum_{n=0}^{+\infty}\sum_{l=-n}^{n}(-1)^n I_n^{-l}(\mathbf{X}')O_n^l(\mathbf{X}).$$

*The next theorem is used to establish M2M (Outer-to-Outer) and M2L (Outer-to-Inner) operators.*

**Theorem 2** (Outer-to-Outer, Outer-to-Inner Laplace translation theorem). *Let* $\|\mathbf{X}\| > \|\mathbf{X}'\|$, *then*

$$O_n^l(\mathbf{X}-\mathbf{X}') = \sum_{j=0}^{+\infty}\sum_{k=-j}^{j}(-1)^j I_j^{-k}(\mathbf{X}')O_{n+j}^{l+k}(\mathbf{X}) \quad \forall (n,l)\in\mathbb{N}\times\mathbb{Z} \quad \textit{with } |l|\leqslant n.$$

*The last theorem corresponds to the Third Addition Theorem in* [2], *and it is used for L2L (Inner-to-Inner) operator.*

**Theorem 3** (Inner-to-Inner Laplace translation theorem)

$$I_n^l(\mathbf{X}-\mathbf{X}') = \sum_{j=0}^{n}\sum_{k=-j}^{j}(-1)^j I_j^{k}(\mathbf{X}')I_{n-j}^{l-k}(\mathbf{X}) \quad \forall (n,l)\in\mathbb{N}\times\mathbb{Z} \quad \textit{with } |l|\leqslant n.$$

*Thus, according to the classical translation theorem and the Outer-to-Inner Laplace translation theorem, we have the following expression for the M2L operator that converts multipole expansion terms into local expansion terms, as defined in* [22].

**Proposition 1** (M2L operator). *Let* $M_n^l$, *with* $n \geqslant 0$, $|l| \leqslant n$, *the multipole expansion terms centered in* $\mathbf{z}_1$, *the local expansion terms* $L_j^k$ *centered in* $\mathbf{z_2}$ *write*

$$L_j^k = \sum_{n=0}^{+\infty}\sum_{l=-n}^{n} M_n^l O_{j+n}^{-k-l}(\rho,\alpha,\beta) \quad \forall (j,k)\in\mathbb{N}\times\mathbb{Z} \quad \textit{with } |k|\leqslant j,$$

*where* $(\rho,\alpha,\beta)$ *are the spherical coordinates of the M2L vector* $\mathbf{z_2}-\mathbf{z_1}$.

The $O_{j+n}^{-k-l}(\rho,\alpha,\beta)$ are named the *M2L transfer functions*. The implementation of this formulae is named hereafter the *classic M2L*. Finally, we emphasize the following property which is also valid for the Outer and Inner functions, as well as for the multipole and local expansion terms: this enables the computation of only terms with positive orders in multipole and local expansions.

**Lemma 1** (Symmetry among orders). $Y_n^{-l} = \overline{Y}_n^l$ *where* $\bar{z}$ *denotes the complex conjugate of* $z\in\mathbb{C}$.

## 2.2. Error bound analysis

As exposed in Section 1, the *M2L* operator concretely uses finite multipole expansion to compute (finite) local expansions. When denoting $P$ as the maximum degree of the expansions, whose terms have degrees therefore ranging from 0 to $P$, two different kinds of *M2L* expressions can then be used.

The first one, named *M2L kernel*[3] *with double height*, corresponds to the outer sum on $n$, in the expression of the *M2L* operator given in Proposition 1, stopping at $n = P$, since in this case the $O_j^k$ terms range up to $2P$. This one is natural and commonly used, but even if some interesting works have been done to estimate the behavior of the error induced by such *M2L* expression (see [19,21]), no sharp error bound has been found yet.

In the second *M2L* expression the outer sum on $n$ stops at $n = P - j$, so that the maximum degree of the $O_j^k$ used is $P$. This is named *M2L kernel with single height*. This has been used for example in the DPMTA (Dis-

[3] The *kernel* name has been inspired by [3].

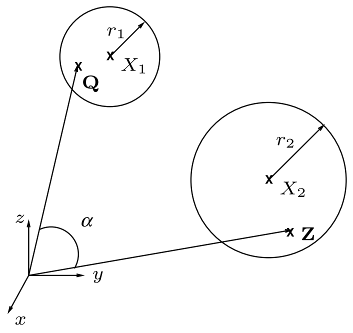


Fig. 1. Our problem configuration.

tributed Parallel Multipole Tree Algorithm) code developed at Duke University but the proof in the error bound given in [3] is wrong.[4] We however recommend the reading of this appendix since it presents a worst-case error analysis similar to the one used hereafter. In particular it shows that no additional error is introduced by the *M2M* operation. As explained in [2] the *L2L* operation does not introduce any error at all. We can thus focus on the *M2L* operation.

In the following we present a sharp error bound (similar to the one presented in [3]) respected by an *M2L* operator with single height kernel. It has to be noted that this error bound for single height *M2L* kernel was briefly presented by White & Head-Gordon [20], but they have used double height kernel in their implementation.

In order to introduce this error bound, we study the potential in point $\mathbf{Z}$ due to one single unit charge located in $\mathbf{Q}$ as pictured in Fig. 1. We denote by $\mathcal{B}_1$ (respectively $\mathcal{B}_2$) the ball centered in $\mathbf{X_1}$ with radius $r_1$ (respectively $X_2$ and $r_2$). $\mathbf{Q}$ is enclosed in $\mathcal{B}_1$, $\mathbf{Z}$ in $\mathcal{B}_2$. Moreover, denoting $R = \|\mathbf{X_1} - \mathbf{X_2}\|$, we assume that $R > r_1 + r_2$.

We first proceed as in [2]. Let $\alpha$ the angle between "vectors" $\mathbf{Q}$ and $\mathbf{Z}$, $r_>$ (respectively $r_<$) the maximum (respectively minimum) between the norm of $\mathbf{Q}$ and the one of $\mathbf{Z}$, and $P_n$ the Legendre polynomials, we have

$$\Phi(\mathbf{Z}) = \frac{1}{\|\mathbf{Z} - \mathbf{Q}\|} = \sum_{n=0}^{+\infty} \frac{r_<^n}{r_>^{n+1}} P_n(\cos\alpha). \tag{4}$$

Defining $\Phi_P(\mathbf{Z})$ the potential with finite summation up to $P$, we have, since for all $x \in [-1, 1]$, $|P_n(x)| \leqslant 1$,

$$\|\Phi(\mathbf{Z}) - \Phi_P(\mathbf{Z})\| \leqslant \frac{1}{r_> - r_<} \left(\frac{r_<}{r_>}\right)^{P+1}. \tag{5}$$

This leads to the following error bound

**Proposition 2.** $\forall \mathbf{Q} \in \mathcal{B}_1$, *and* $\forall \mathbf{Z} \in \mathcal{B}_2$, *we have*, *under the condition* $R > r_1 + r_2$

$$\|\Phi(\mathbf{Z}) - \Phi_P(\mathbf{Z})\| \leqslant \frac{1}{R - (r_1 + r_2)} \left(\frac{r_1 + r_2}{R}\right)^{P+1}.$$

**Proof.** Since $\|\mathbf{Z} - \mathbf{Q}\| = \|((\mathbf{Z} - \mathbf{X_2}) + (\mathbf{X_1} - \mathbf{Q})) + (\mathbf{X_2} - \mathbf{X_1})\|$ and since $\|(\mathbf{Z} - \mathbf{X_2}) + (\mathbf{X_1} - \mathbf{Q})\| \leqslant r_1 + r_2$, this results directly from the inequality (5). □

We can now decompose the potential in the same way the FMM does while maintaining this error bound, and we obtain the following theorem (whose proof is given in Appendix A).

**Theorem 4.** *The M2L operator* **with single height *M2L* kernel** *strictly respects the error bound in Proposition 2.*

[4] Eq. (C.23) in Appendix C of [3] is wrong because the composition of differential operators differs from the product of the derivatives.

Since the error in Proposition 2 is higher than the one due to the multipole expansion terms (see [11]), and since no additional error is introduced by the *M2M* and *L2L* operators, the error bound of the FMM with a single height kernel *M2L* operator is therefore the same as in Proposition 2.

It has to be noted that the condition of well-separateness, with either $ws = 1$ or $ws = 2$, is in fact more strict than $R > r_1 + r_2$ in order to ensure a fast enough convergence. With $ws = 1$ for example, we have, for a cell of side $l$: $r_1 = r_2 = \frac{\sqrt{3}}{2}$ and $R \geqslant 2$. Thus

$$\|\Phi(\mathbf{Z}) - \Phi_P(\mathbf{Z})\| \leqslant \frac{1}{R-(r_1+r_2)}\left(\frac{\sqrt{3}}{2}\right)^{P+1} = \frac{1}{R-(r_1+r_2)}(0.866)^{P+1}.$$

As a result, this error bound can safely be used in the case of single height kernel in order to determine the value of $P$ according to the required precision. Moreover since the double height kernel, compared to the single height kernel, only adds terms in the *M2L* formula, it respects at least this error bound but even adds precision with additional cost at runtime. The problem is that we cannot predict this gain in the accuracy of the FMM, and therefore we cannot compare single and double height kernels according to the tradeoff between theoretical accuracy and runtime. Only comparisons with practical accuracies will be possible as described in Section 4.

We will now present, for both single and double kernel heights, several current improvements of the *M2L* operator that reduce its theoretical operation count.

## 2.3. Fast Fourier transform

The use of fast Fourier transform (FFT) in order to speed up the *M2L* computation has been implemented by Elliott and Board [3,8], based on a method originally introduced by Greengard and Rokhlin [9]. This work was performed only for single height *M2L* kernel and the block version used to prevent numerical instability as written in DPMTA has restricted the possible values of $P$ to multiples of the block size. In [22] we have generalized this FFT improvement to both single and double height *M2L* kernels and our block version can handle any value for $P$.

The principle behind the use of FFT in FMM is to view the *M2L* operator as one $2D$ convolution (or correlation) between two sequences. This convolution is faster computed thanks to one forward $2D$ FFT, one point-wise product and one backward $2D$ FFT, *provided that the two sequences are periodic*. As presented in [23] for *M2M*, we have to use additional null terms in order to build periodic sequences out of the multipole and local expansions and the *M2L* transfer function: this process is named *zero-padding*. However, due to large varying magnitude in the norm of the expansion terms, this FFT computation results in numerical instabilities when $P$ grows. This issue has partly been resolved by two methods: the *block* FFT and the *polynomial scaling* scheme.

### 2.3.1. Block fast Fourier transform

The *block* FFT [8] is based on a decomposition of the expansion arrays along the degree dimension, which implies a *large-grain convolution* among the blocks for the point-wise product. More details can be found in [22] and in our implementation we have used FFTW [24] as an efficient FFT implementation.

As for operation counts the most time-consuming part of the FFT scheme is the point-wise product. While each point-wise product runs in $\mathcal{O}(P^2)$ in the non-block version, the large-grain convolution of the block version leads to an operation count in $\mathcal{O}(P^3)$. This part of the computation does not match any of the standard BLAS calls (see Section 3) and offers no data reuse: no special speedup can be expected from its implementation.

The block version does not however ensure complete numerical stability as shown in Fig. 2(a), where we use the $L_2$ norm as practical error measurement:

$$\varepsilon_{L_2} = \left(\frac{\sum_{i=1}^{M}(\Phi_{\mathrm{Dir}}(\mathbf{X}_i) - \Phi_{\mathrm{FMM}}(\mathbf{X}_i))^2}{\sum_{i=1}^{M}\Phi_{\mathrm{Dir}}(\mathbf{X}_i)^2}\right)^{\frac{1}{2}}.$$

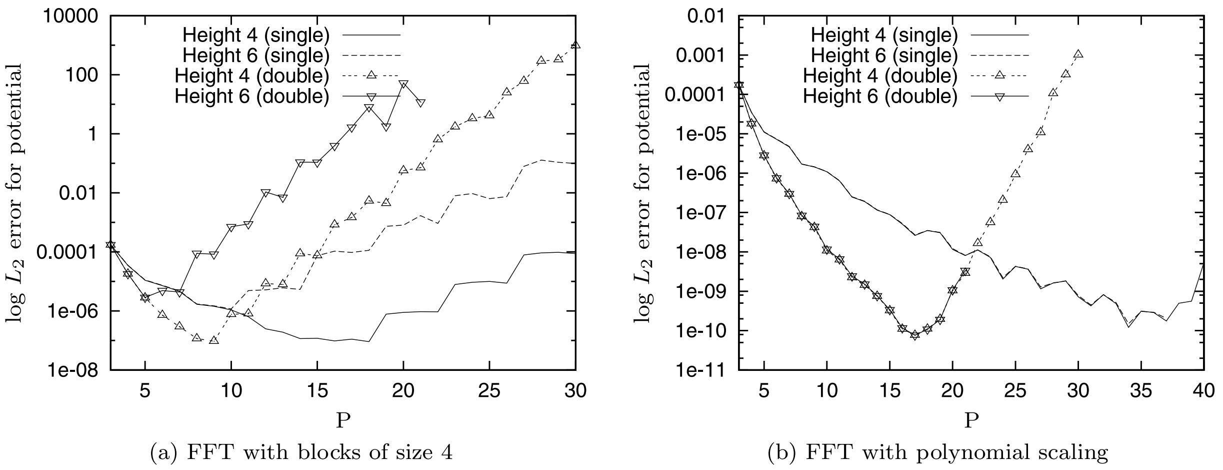


Fig. 2. Logarithmic $L_2$ error for the potential according to $P$ for both single and double height $M2L$ kernels. Tests performed on 100,000 particles, uniformly distributed, with octrees of height 4 and 6. All particle coordinates are inside [0, 1.0]. Tests run on IBM Power4 with double precision.

More precisely, the direct computation is used to compute the exact potential $\Phi_{\mathrm{Dir}}$ of $M = 1000$ bodies (chosen randomly among the $N$ bodies), and the error generated for these $M$ bodies is then computed according to the $L_2$ norm. It can be noted that the error on each force coordinate ($fx$, $fy$ and $fz$) presents numerical unstabilities for the same $P$ values as the potential error here plotted. For growing values of $P$ these numerical instabilities may be explained by the influence of the orders in the magnitude of the norm of the terms used: the block version is indeed only applied in the degree dimension. But numerical instabilities are also subject to the octree height used: an increasing height indeed decreases the distance between two cell centers in the $M2L$ transfer function terms, as well as the distance between a cell center and a particle location in the multipole expansion terms. Numerical instabilities hence arise when the order of magnitude of these distances is becoming too small. Of course this behavior depends on the size of the computational box that encloses all particles, but with a bigger computational box the numerical instabilities would nevertheless appear for greater heights.

These instabilities can however be reduced with lower block size: while, for single height $M2L$ kernel and octree height 4, the FFT with block size 4 becomes unreliable for $P$ greater than 14 (see Fig. 2(a)), a FFT with block size 3 is stable until 27, but is of course slower to compute because of the higher operation count due to the higher number of blocks. When running DPMTA (version 3.1.2p3) the instabilities appear for the same $P$ values and octree heights. We have also compared in [22] the runtime of the FFT improvement in DPMTA (with fixed interaction list with $ws = 1$) and in our FMB code: similar CPU times were measured which validates the efficiency of our implementation.

With double height $M2L$ kernel, the maximum degree of the $O_j^k$ is $2P$: the sizes of the arrays used in the FFT and in the point-wise product are thus roughly multiplicated by 4 (possibly thanks to zero-padding). But since $O_j^k$ have now degrees up to $2P$, the range of magnitude is even greater and they become unstable for even lower values of $P$: the FFT with double height kernel requires therefore even more the block version. The instabilities in the block FFT, as in the non-block version, appear however for lower $P$ values than in the single height case: see Fig. 2(a). Finally as for operation counts, we focus in the block FFT on the large-grain convolution which is the most time-consuming part: the ratio between single and double height kernels is here 8, which clearly does not favor the double height kernel compared to the ratio of 6 in classic $M2L$ computation.[5]

[5] For classic $M2L$ computation with single height $M2L$ kernel, we have thanks to Lemma 1 an operation count like: $\frac{1}{12}(P+3)(P+1)(P+2)^2$. With double height kernel, we have: $\frac{1}{2}(P+2)(P+1)^3$. Therefore the ratio among the two heights is roughly 6.

### 2.3.2. Fast Fourier transform with polynomial scaling

In the technical report [9], Greengard and Rokhlin have introduced another scheme to postpone the numerical instabilities in the 2D FMM: the polynomial scaling. The principle is to scale all the terms with polynomial factors, before and after the *M2L* computation with FFT, in order to reduce the magnitude of their norm.[6] More precisely, at the octree level $h$ we rewrite the *M2L* operator defined in Proposition 1 (here, with double height kernel) as

$$\frac{L_j^k}{(s_h)^j} = \sum_{n=0}^{P} \sum_{l=-n}^{n} M_n^l (s_h)^n \frac{O_{j+n}^{-k-l}}{(s_h)^{n+j}},$$

where $s_h$ is the scaling at level $h$. According to [9] we choose $r \cdot s_h = 2P$, where $r$ is the norm of the *M2L* vector.[7] With $ws = 1$, we have $2d_h \leqslant r \leqslant 3\sqrt{3}d_h$, where $d_h$ is the cell side length at level $h$. In practice the value that gives the more stable computations appears to be $r = 4d_h$. We have therefore: $s_h = \frac{P}{2d_h}$.

After this scaling on multipole expansions and on *M2L* transfer functions, the non-block version of the FFT proceeds normally and the local expansions terms are then unscaled before moving to the next octree level of the downward pass. We thus keep the $\mathcal{O}(P^2)$ point-wise product of the non-block version, which results in a $\mathcal{O}(P^2 \log P)$ operation count for the FFT *M2L* computation scheme with polynomial scaling.

Fig. 2(b) shows the numerical stability of our polynomial scaling implementation. Thanks to this polynomial scaling, the numerical instabilities are not anymore subject to the octree height, and the computation scheme is more stable for growing values of $P$. However, as mentioned in [9] the factorial terms will always grow larger than the polynomial ones, which results in numerical instabilities for high $P$ values (greater than 34 for single height *M2L* kernel and greater than 17 for double height *M2L* kernel, in Fig. 2(b)). This remaining instability may explain why this computation scheme has not been used in the literature since the technical report [9] (except its reference in [8]).

## 2.4. Rotations

The use of *rotations* has already been introduced in several articles: see [10–12,15,25]. This improvement enables the computation of the *M2L* operator[8] in $\mathcal{O}(P^3)$, against $\mathcal{O}(P^4)$ for their classic version. We have chosen to use the formulae detailed by Gumerov and Duraiswami in [15] since: they use the same definition for spherical harmonics, they use symmetries to speed up the computation and they focus only on the needed rotations. Moreover the recurrence is performed on real numbers, and not complex ones, and is simple to initiate. However no proof is given on the numerical stability of the formulae used.

The improvement in the use of rotations for *M2L* operator is based on the fact that the cost of an *M2L* operation performed along the **z** axis is $\mathcal{O}(P^3)$ against $\mathcal{O}(P^4)$ for general *M2L*. This is due to a property of the associated Legendre functions and this is valid for both single and double height *M2L* kernels. For a general *M2L* operation whose *M2L* vector is not aligned with the **z** axis, we have first to rotate the Cartesian coordinate system so that the *M2L* vector is along the **z** axis, then we perform the *M2L* operation, and finally we rotate back the coordinate system. The whole procedure is worthwhile since the first rotation on multipole expansion and the second on local expansion are both performed with $\mathcal{O}(P^3)$ operations. These rotations are performed thanks to *rotation coefficients* applied to the spherical harmonics and precomputed at each level of the octree during the downward pass. More details about the implementation of this scheme can be found in [22].

The numerical stability of this scheme has been checked through whole FMM computations: no difference was detected in [22] between the classic *M2L* scheme and the *M2L* scheme with rotations, for values of $P$ up to 30, for both single and double height kernels and for several octree heights. When considering detailed operation count the ratio between single and double heights for *M2L* kernel appears to be roughly $\frac{20}{7}$ which favors the double height kernel compared to the ratio of 6 of the classic *M2L* implementation (see Footnote 5).

[6] We also refer the reader to [8,22] for a simple scaling without polynomial terms.

[7] In [9], $r.s_h = P$ was used in the $2D$ FMM, but here in the $3D$ FMM the factorial terms grows mainly as $(2P)$!

[8] This applies also to *M2M* and *L2L* operators.

Moreover, the memory requirements are very low since the number of rotation coefficient arrays equals the number of *M2L* transfer functions (316 with $ws = 1$).

In [22] we have also discussed the possibility of introducing the BLAS routines (presented in Section 3) in order to speed up the *M2L* computation with rotations, but no efficient solution was found: while the rotations of multipole and local expansions cannot be written as matrix–vector products, the matrix–vector product corresponding to the *M2L* operation along the $\mathbf{z}$ axis computes too many useless terms and cannot be extended to matrix–matrix products.

### 2.5. Plane waves

Another computation scheme for the *M2L* operator is based on plane wave expansions. It has been presented in [11] and improved in [12]. We now briefly present this scheme, and we refer the reader to these articles for more details.

The plane wave scheme is based on the following rewriting of the potential between two points $\mathbf{X} = (x, y, z)$ and $\mathbf{X}' = (x', y', z')$, with $z > z'$:

$$\frac{1}{\|\mathbf{X} - \mathbf{X}'\|} = \frac{1}{2\pi} \int_0^\infty \mathrm{e}^{-\lambda(z-z')} \int_0^{2\pi} \mathrm{e}^{\mathrm{i}\lambda((x-x')\cos\alpha + (y-y')\sin\alpha)}\, \mathrm{d}\alpha\, \mathrm{d}\lambda,$$

where the double integral is estimated by quadrature formulae that depends on the required precision. Four precisions are used in [12] (see Section 4.5 for details), and new quadrature formulae have to be written for each new precision. These precisions correspond to theoretical error bounds ensured by the FMM. More details about the quadrature formulae can be found in [26,27].

In order to compute a given *M2L* operation, the source multipole expansion is first converted into a plane wave expansion thanks to the quadrature formulae. The plane wave expansion is then translated from the center of the multipole expansion to the center of the target local expansion, where it is finally converted into a local expansion. This is however valid only if the error introduced by the plane wave expansions corresponds to the error due to multipole and local expansions. The number of quadrature nodes and plane waves is thus tied up to $P$ and $P^2$ (see [12]). The translation of the plane wave expansion is then performed in $\mathcal{O}(P^2)$, which justifies the interest of this computation scheme, and the total operation count for *M2L* operation grows like $\mathcal{O}(P^3) + \mathcal{O}(P^2)$ (due to conversions and possible additional rotations).

In summary of Section 2, we have shown that the single height *M2L* kernel, contrary to the double height one, respects the FMM error bound which justifies its implementation. The $\mathcal{O}(P^3)$ block FFT, the $\mathcal{O}(P^2 \log P)$ FFT with polynomial scaling and the $\mathcal{O}(P^3)$ rotation scheme reduce the operation count of the *M2L* computation, but the two FFT enhancements, contrary to the use of rotations, require important extra memory storage and may result in unpredictable numerical instabilities. In addition, the plane wave scheme is stable for all $P$ values and presents a low $\mathcal{O}(P^3) + \mathcal{O}(P^2)$ operation count. We will now propose an alternative to speed up the *M2L* computation.

## 3. Multipole to local operator: implementation with BLAS routines

The basic linear algebra subprograms (BLAS) (see [16,28,29]) are a standard interface for some usual linear algebra operations such as a dot product of 2 vectors (level 1 BLAS), a matrix–vector product (level 2 BLAS) or a matrix–matrix product (level 3 BLAS). Optimized for the pipelines of the floating point units and for the hierarchical memory of the computer, the BLAS routines offer an efficient implementation of these operations, and the higher the level of the BLAS used, the higher the speedup they reach.

BLAS routines have already been used for hierarchical $\mathcal{O}(N)$ $N$-body algorithms by Hu and Johnsson [30] with Anderson's method [31] which uses different expansions than the FMM, and hence translation/conversion operators, but has the same algorithm for the upward and downward passes. Here we propose the first BLAS implementation for the *M2L* operator of the FMM for both single and double height kernels. In order to achieve the highest efficiency, we also detail a scheme with *recopies* that enables to use level 3 BLAS routines, and we show how to avoid these *recopies* thanks to new data storages in memory.

While the original FMM formulae of Greengard and Rokhlin (see [2]) do not enable a rewriting of their corresponding operators ($M2M$, $M2L$ or $L2L$) as matrix–vector products, this can be easily done with the simpler formulae of [3,23] or [20]. The full FMM algorithm has also been rewritten in terms of matrix operations in [32]: here, we only focus on the rewriting of the $M2L$ operator as a matrix–vector product.

As we compute only terms with positive orders in the local expansion, we can write each $M2L$ operation as the following matrix–vector product, here written for $P = 2$ with single height $M2L$ kernel:

$$\left[\begin{array}{c} L_0^0 \\ \hline L_1^0 \\ L_1^1 \\ \hline L_2^0 \\ L_2^1 \\ L_2^2 \end{array}\right] = \left[\begin{array}{c|ccc|ccccc} O_0^0 & O_1^1 & O_1^0 & O_1^{-1} & O_2^2 & O_2^1 & O_2^0 & O_2^{-1} & O_2^{-2} \\ \hline O_1^0 & O_2^1 & O_2^0 & O_2^{-1} & 0 & 0 & 0 & 0 & 0 \\ O_1^{-1} & O_2^0 & O_2^{-1} & O_2^{-2} & 0 & 0 & 0 & 0 & 0 \\ \hline O_2^0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ O_2^{-1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ O_2^{-2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \cdot \left[\begin{array}{c} M_0^0 \\ \hline M_1^{-1} \\ M_1^0 \\ M_1^1 \\ \hline M_2^{-2} \\ M_2^{-1} \\ M_2^0 \\ M_2^1 \\ M_2^2 \end{array}\right]. \tag{6}$$

The computed vector is named *local vector*. The matrix is named *M2L transfer matrix* and denoted $\mathbf{T}_{M2L}$. With double height $M2L$ kernel the matrix is dense: terms $O_j^k$ with $j > P$ do not vanish as in the single height case. The building of this matrix, detailed in [22], is rather straightforward. The vector used is named *multipole vector*: terms with negative orders have to be stored.

## 3.1. Implementation with level 2 BLAS routines

Our matrices are stored in column storage mode, and we use the transfer matrix in its transposed form: it is indeed generally more efficient to compute $C \leftarrow A^T \cdot B$ in row storage mode for $A$, than $C \leftarrow A \cdot B$ with column storage mode for $A$; indeed, when considering the BLAS implementation (see [33] for a default implementation), the first solution leads to less writings, with however more readings, than the second one.

### 3.1.1. Double height M2L kernel

The matrix $\mathbf{T}_{M2L}$ is dense: the use of level 2 BLAS *ZGEMV* routine is therefore obvious, and this routine will automatically optimize the computation of the matrix–vector product to the underlying superscalar architecture. In order to differentiate it from the other BLAS method that will be used later, we name it *full_blas*.

### 3.1.2. Single height M2L kernel

The matrix $\mathbf{T}_{M2L}$ is now sparse: we therefore have to split the sparse matrix–vector product in several dense block products.

We refer the reader to the BLAS literature (see [34] for example) for the underlying techniques used to fill at best the pipelines of the floating point execution units in order to reach peak performance of the processor. These techniques are mainly: loop ordering for best spatial and temporal locality among data, loop blocking in order to provide maximum cache reuse, temporary copies in local arrays for problems with "leading dimensions" of the matrices (see BLAS routine interfaces), loop unrolling, register blocking, data prefetch, etc.

In [35], it has been shown that level 3 BLAS routines can efficiently be implemented only with the *GEMM* level 3 BLAS routine and a few level 2 BLAS routines. For portability purpose, as well as for simplicity, we prefer to adopt such approach. We will thus decompose the sparse matrix–vector product corresponding to $M2L$ operator in several *ZGEMV* block products in the most efficient way. *ZGEMV* routine is used here since we treat a matrix–vector product, but in Section 3.2 we will see how to use matrix–matrix products, and *ZGEMM* routine will then be used as in [35]. All the optimizations recalled above will be used but left as much as possible to the underlying (and machine dependent) *ZGEMV*/*ZGEMM* BLAS routine called. We point out now one important fact: in our implementation of the FMM, we are free in the memory storage of the blocks of $\mathbf{T}_{M2L}$ since its construction is precomputed at each level of the octree in the downward pass of the FMM.

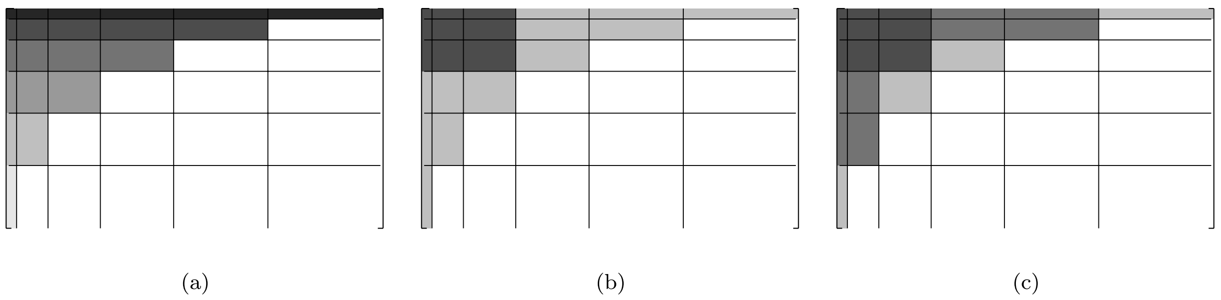


Fig. 3. *block_blas* decomposition for $P = 5$. (a) Decomposition with only strips: the different gray values show the different strips. (b) The "biggest upper left square" sub-matrix of $\mathbf{T}_{M2L}$ for $P = 5$ is pictured with the darker gray value. (c) In the remaining sub-matrices "on the right" and "below", whose sub-blocks are disposed in the same way as $\mathbf{T}_{M2L}$ for $P = 2$ (see Eq. (6)), we can recursively isolate a "biggest upper left square" sub-matrix.

Therefore, most of the problems that arise when considering the leading dimension of a matrix will be irrelevant with our blocks. With these principles in mind, we will present an efficient block decomposition, named *block_blas*, for the matrix $\mathbf{T}_{M2L}$ with single height *M2L* kernel.

In [35], triangular matrix–matrix product was performed with *ZGEMM* routines thanks to a decomposition of the triangular matrix in *strips*. First, we proceed similarly using strips in the horizontal direction since the transfer matrix is stored by rows, and since this leads to several traversals of the multipole vector (one for each strip) against one single traversal on the local vector. In other words, with strips in the row direction, the resulting blocks of the local vector are updated one after the other, whereas the corresponding data of the multipole vector may be reloaded several times during the block matrix–vector product. As the local vector is traversed for updating (i.e. both reading and writing) and the multipole expansion for reading, this is more efficient than vertical strips which lead to the contrary. In practice, our strips strictly[9] respect the underlying structure of $\mathbf{T}_{M2L}$ as pictured on Fig. 3(a): each strip is separately stored and separately computed with one dedicated call to *ZGEMV* routine. One obvious problem is that, as *P* grows, the first strips are too thin and too long while the last ones are too thick and too short: this can prevent the BLAS routine to internally decompose those strips according to the hierarchical memory of the computer.

But the underlying structure of $\mathbf{T}_{M2L}$ can be considered as recursive for a given *P*. Indeed when isolating the "biggest upper left square" (in the number of underlying sub-blocks) sub-matrix, with half of the sub-blocks in both the row and the column dimensions, as pictured in Fig. 3(b) and (c), we are left with one sub-matrix "on the right" and another one "below" whose sub-blocks are disposed in the same way as $\mathbf{T}_{M2L}$ for $\lfloor \frac{P}{2} \rfloor$. Detailed expression of this recursion can be found in [22].

Hence we use as much as possible dense matrices (i.e. the upper left sub-matrices) which are efficiently treated by *ZGEMV* routine. With small values for *P*, which corresponds also to the terminal cases of the recursion, it may be not worth isolating this dense matrix: in these cases we use our strips. The inefficiency due to the shape of the first and the last strips is therefore minimized since we reserve the strips for the terminal cases. The terminal value for the recursion, hereafter named $s_{\text{block}}$, will have to be determined experimentally.

## 3.2. Level 3 BLAS routines

Since matrix–matrix–matrix product requires $\mathcal{O}(N^2)$ memory storage relatively to $\mathcal{O}(N^3)$ operation count, it is easier to overlap memory latency with computation of the floating point execution units with level 3 BLAS than with level 2 BLAS, and thus to reach peak performance of the processor. We will therefore try to group multiple *M2L* operations in one single matrix–matrix product.

During the downward pass, at a given level of the octree, all *M2L* operations that have the same *M2L* vector (see Proposition 1) share the same *M2L* transfer matrix. When considering all pairs of multipole and local expansions that share the same *M2L* vector, it is thus possible to concatenate all their multipole vectors as

[9] Another decomposition that uses additional zeros to obtain strips with optimal number of rows is discussed in [22].

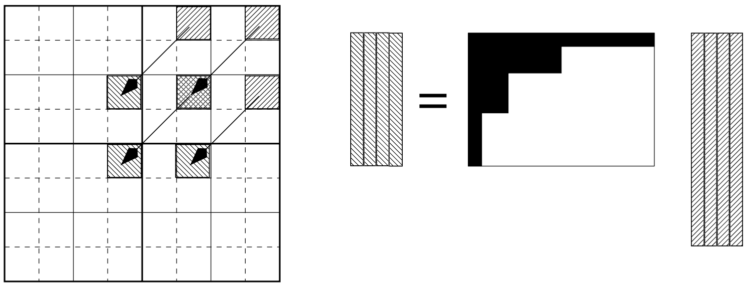


Fig. 4. Concatenation of multipole and local expansions in order to use level 3 BLAS. The 4 *M2L* operations that share the same *M2L* vector are computed together in one single matrix–matrix product: the corresponding multipole and local vectors are hence concatenated in one multipole matrix and in one local matrix, both with four columns. The transfer matrix represented here is for a single height *M2L* kernel.

columns of one single *multipole matrix* $\mathbf{M}^M$. The local vectors are also concatenated according to the same order to form one single *local matrix* $\mathbf{M}^L$. Then the matrix–matrix product $\mathbf{M}^L = \mathbf{T}_{M2L} \cdot \mathbf{M}^M$ computes the corresponding *M2L* operations all at once as described in Fig. 4. The concatenation is easily achieved thanks to the column storage of our matrices.

First, we will roughly consider that the best efficiency is obtained with level 3 BLAS when the maximum number of multipole and local expansions are concatenated each time (see Section 3.2.3 for revisions of this assertion). In the following, we will thus see how to concatenate the maximum number of multipole and local expansions for one given *M2L* transfer matrix, and we will then present the implementation of this matrix–matrix product thanks to level 3 BLAS calls.

With free-space boundary conditions[10] (FBC), where all the space outside of the computational box is considered as empty, the cells located at the boundaries of the computational box, in each level, have an incomplete interaction list with less than 189 members for a *well-separateness* criterion $ws = 1$ (see Section 2.1). In order to ease their computation, and for efficiency purpose, these cells with incomplete interaction list are computed separately when using level 3 BLAS for *M2L*.

### 3.2.1. Computing the local expansions of cells with incomplete interaction list

With free-space boundary conditions, the "cells with incomplete interaction list" are all the cells whose parent have at least one neighbor that is outside of the octree. In order to treat all the *M2L* operations for the local expansions of such cells, we use copies of the multipole and local vectors. The *M2L* computation is thus performed as $\mathbf{M}^L \leftarrow \mathbf{T}_{M2L} \cdot \mathbf{M}^M + \mathbf{M}^L$ and the columns of the local matrix are then *recopied* in the original local expansions.

For a given *ws* value, the *M2L* vectors of the interaction list of a given cell are determined according to the *type of child* of the cell which describes the location of the cell center relatively to the center of its father. In 3D, there are eight different possible *types of child*. Some *M2L* vectors, but not all of them, are shared by all the *types of child*. We first loop on each possible *M2L* vector and secondly, inside this first loop, we loop on each *type of child* that matches the current *M2L* vector. This has been preferred to the opposite ordering of these two loops since it leads to a greater number of vectors treated per matrix–matrix product.

### 3.2.2. Computing the local expansions of cells with complete interaction list

All cells with complete interaction list have the same interaction list size, and all cells of the same *type of child* share exactly the same *M2L* vectors. This regularity enables well-suited algorithms.

The most simple uses each time *recopies* as in [30] in order to treat altogether the maximum number of pairs of multipole and local expansions. Contrary to the cells with incomplete interaction list, we first loop on each *type of child* and secondly loop on each *M2L* vector corresponding to the current *type of child*: see Algorithm 1.

[10] Such specificity does not appear with "Periodic Boundary Conditions", where the computational box is periodically replicated in each dimension.

**Algorithm 1** Scheme with *recopies*. $\mathcal{V}_{M2L}$ denotes the set of all possible *M2L* vectors, $\mathcal{T}_C$ the set of all different *types of child* and $\mathbf{T}_{M2L}(v)$ the *M2L* transfer matrix corresponding to the *M2L* vector $v$.

```
1: for all $t \in \mathcal{T}_C$
2:    Copy all local vectors in $\mathbf{M}^L$;
3:    for all $v \in \mathcal{V}_{M2L}$ corresponding to $t$
4:       Copy the corresponding multipole vectors in $\mathbf{M}^M$;
5:       Perform $\mathbf{M}^L = \mathbf{T}_{M2L} \cdot \mathbf{M}^M$;
6:    end for
7:    Copy back all the local vectors;
8: end for
```

Thanks to the regularity, the *recopies* of the local expansions can be here performed out of the second loop, and the opposite order of the loops would lead to more copies for the local vectors while having the same number of vectors treated each time.

It is however possible to avoid the additional cost of *recopies* thanks to special *data storages* of our multipole and local vectors which are obtained through a "rearrangement" of these vectors in memory performed before the downward pass. In the first data storage, named *row* storage and illustrated in Fig. 5, we store consecutively in memory all the local vectors that belong to cells of the same *type of child* along a row in one given dimension of the 3D space. This is done for each row of the level, and the same storage is also done for the multipole vectors. Note however that for local vectors, only cells with complete interaction list have their local expansions rearranged in rows, while for the multipole expansions the rearrangement has to be performed for all cells of the level. With such data storage, we can call a level 3 BLAS routine starting at the local vector of the first cell of each row, with the corresponding *M2L* transfer matrix and the corresponding multipole vector. It can be noted that Morton ordering (see Section 2.1) is used here in order to access cells according to the coordinates of their center. With FBC, at level $l$ and for a given *type of child*, the local expansions are hence rearranged in $(2^{l-1} - 2 \cdot ws)^2$ rows of size $2^{l-1} - 2 \cdot ws$.

The size of the rows corresponds to the number of local vectors treated per level 3 BLAS call; this might be not enough to obtain the best efficiency: for example there is only six columns at level 4. That is why we propose a second data storage, named *slice* storage, that stores consecutively the rows in memory in order to form a *slice* and thus enables several rows to be treated with one single level 3 BLAS call. In order to have a correct correspondence between slices of local expansions and slices of multipole expansions, we have to insert *blank*

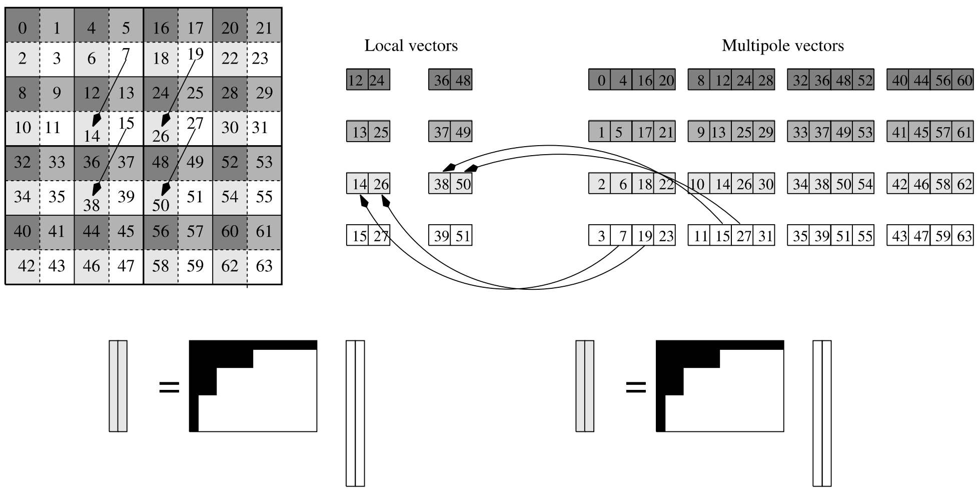
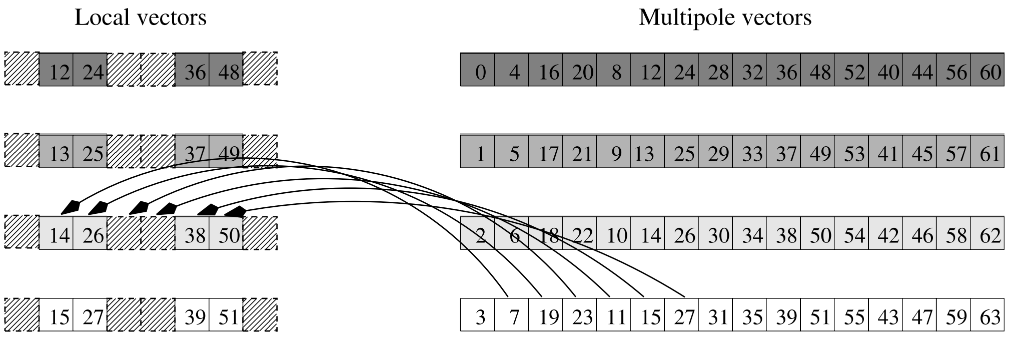


Fig. 5. *row* storage (2*D*, $ws = 1$, level 3). The cells are indexed according to Morton ordering. Each of the four *types of child* has a different gray value. The expansions of the cells with same *type of child* along a given row are concatenated: the four *M2L* operations, whose *M2L* vectors are represented on the quadtree, can then be directly computed with two matrix–matrix products (level 3 BLAS) without *recopies*.



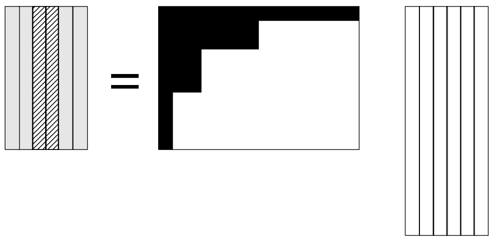


Fig. 6. *slice* storage (2*D*, $ws = 1$, level 3): see also Fig. 5. The rows of the *row* data storage mode are concatenated with *blank boxes* (marked with stripes) in between: the matrix–matrix product now involves multipole and local matrices with six columns, against two columns with *row* storage (see Fig. 5).

*boxes* between rows of local expansions. These *blank boxes* correspond to cells not belonging to the octree whose local vectors are computed uselessly. They are actually required to compensate the absence of the cells with incomplete interaction list which are treated separately. With $ws = 1$, there is 1 *blank box* at the beginning and 1 at the end of each row, as pictured in Fig. 6. We can however skip the first *blank box* of the slice (that is to say, the first *blank box* of the first row), and also the last one. With FBC, at level *l*, and for each *type of child*, the local expansions are rearranged in $2^{l-1} - 2 \cdot ws$ slices of size $(2^{l-1} - 2 \cdot ws)^2 + 2 \cdot (2^{l-1} - 2 \cdot ws) - 2$. If *slice* storage enables a better efficiency with the level 3 BLAS, its clear drawback is that it also performs useless extra work due to the *blank boxes*.

This can even be extended to a third data storage, named *level* storage, resulting in concatenation of slices so that the whole level can be computed in one single BLAS call: in this case rows of *blank boxes* have to be added between each slice.

### 3.2.3. Implementation with level 3 BLAS routines

As in Section 3.1.1, the dense matrix–matrix product for double height *M2L* kernel can be directly implemented with one single level 3 BLAS call: the *ZGEMM* routine.

For single height *M2L* kernel the decomposition proposed in Section 3.1.2 is also valid with matrix–matrix products when replacing *ZGEMV* routine by *ZGEMM* one. However, we have to face a new constraint since we have several level 3 BLAS calls per matrix–matrix product: if all the columns of the multipole or local matrix cannot be stored in a level of the hierarchical memory (cache L1, L2 or even L3), or need more memory pages than the TLB (Translation Lookaside Buffer) can address, the whole matrix would have to be reloaded at each BLAS call. A matrix with *NbExp* expansions is therefore split in sub-matrices with a constant number of columns, namely $NbExp_{\max}$, and the same number of rows as the original matrix as described in Fig. 7: each sub-matrix is then treated separately. This decomposition of the matrix–matrix product is hereafter named "$NbExp_{\max}$ decomposition".

Optimal values for $NbExp_{\max}$ will be experimentally determined in Section 3.3.1.

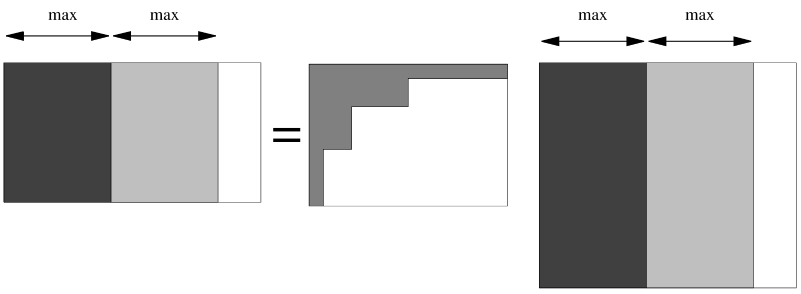


Fig. 7. $NbExp_{\max}$ decomposition (*max* denotes $NbExp_{\max}$). This matrix–matrix product is computed as several matrix–matrix products with at most $NbExp_{\max}$ columns in the multipole and local matrices.

## 3.3. Tests and comparisons

Performance tests have been performed in order to validate the BLAS implementation, along with its parameters, that leads to the fastest *M2L* computation. These tests have been performed either on one IBM Power3-II WH2+(375 MHz, 1.5 GFLOPS, L1 cache size: 64 KB, L2 cache size: 4 MB, and 2 GB of memory) or on one IBM Power4+(1454 MHz, ≈6 GFLOPS, L1 cache size: 64 KB, L2 cache size: 1.41 MB, L3 cache size: 32 MB, and 8 GB of memory), both at LaBRI (Laboratoire Bordelais de Recherche en Informatique, Talence, France). The number of particles used for the simulation is not usually precised since the runtimes here measured depend only on the height of the uniform octree and on $P$, but of course the height used implies a range in the number of particles that balances the near field and the far field computations.

### 3.3.1. Decomposition for single height M2L kernel

In this section, we will see how we have established, for each architecture (Power3 or Power4), the best $NbExp_{\max}$ value, as well as the best $s_{\text{block}}$ value for the *block_blas* routine (see Section 3.1.2). As use of level 3 BLAS calls always improves the performances over level 2 ones (see [22]), these values have been determined when using matrix–matrix products.

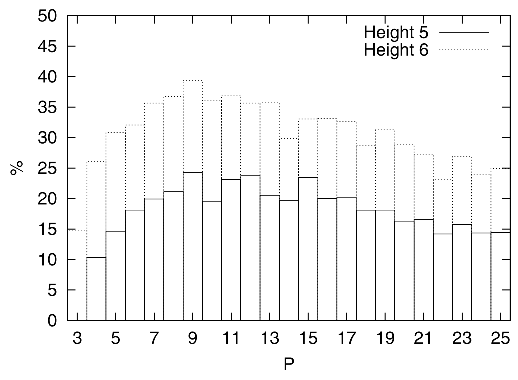
First we need to determine the optimal $NbExp_{\max}$ values. These are clearly machine dependent, and they depend on $P$ too since $P$ determines the number of rows in the multipole matrix and in the local matrix. Depending on the level 3 BLAS scheme used and on the level in the octree, the number of multipole and local expansions concatenated, denoted by *NbExp*, differs significantly. For performance tests with different $NbExp_{\max}$ values, we have only considered *NbExp* values for cells with complete interaction list since this corresponds to the majority of the matrix–matrix products performed. The Table 1 shows *NbExp* values according to our different schemes for cells with complete interaction list.

In practice, tests for optimal $NbExp_{\max}$ values will be performed, according to $P$, for one single “big enough” *NbExp* value. Indeed, as confirmed by more complete tests, higher *NbExp* values will have the same optimal $NbExp_{\max}$ values and lower ones will not need such decomposition. Generally 2744 or 958 are sufficient: see for example Fig. 8. Moreover, results have shown that $NbExp_{\max}$ is mainly independent from $s_{\text{block}}$: when splitting both multipole and local matrices according to $NbExp_{\max}$, the sub-matrices size depends indeed only on $P$ (for the number of rows) and $NbExp_{\max}$ (for the number of columns): see Fig. 7.

Besides, the optimal $s_{\text{block}}$ values, which depend on $P$ and on the machine used, were searched between 1 and $P$. For low values of $P$, the best $s_{\text{block}}$ values equal $P$, which means it is better to use only strips for the

Table 1
*NbExp* values according to our different schemes (with FBC, and $ws = 1$, see Section 3.2.2)

| | Level = 4 | Level = 5 | Level = 6 | Level = 7 |
|---|---|---|---|---|
| *recopies* | 216 | 2744 | 27,000 | 238,328 |
| *row* storage | 6 | 14 | 30 | 62 |
| *slice* storage | 46 | 222 | 958 | 3966 |



Gain in percentage.

| $P$ | $s_{block}$ | $NbExp_{max}$ | $P$ | $s_{block}$ | $NbExp_{max}$ |
|---|---|---|---|---|---|
| 3 | 3 | 921 | 15 | 5 | 102 |
| 4 | 4 | 936 | 16 | 3 | 96 |
| 5 | 5 | 463 | 17 | 3 | 84 |
| 6 | 5 | 403 | 18 | 3 | 84 |
| 7 | 5 | 307 | 19 | 4 | 102 |
| 8 | 3 | 260 | 20 | 7 | 126 |
| 9 | 7 | 210 | 21 | 3 | 120 |
| 10 | 2 | 168 | 22 | 2 | 120 |
| 11 | 10 | 156 | 23 | 5 | 132 |
| 12 | 10 | 126 | 24 | 6 | 120 |
| 13 | 1 | 138 | 25 | 11 | 120 |
| 14 | 1 | 120 | | | |

$s_{block}$ and $NbExp_{max}$ values (Power4).

Fig. 8. Gain in percentage for the full downward pass CPU times offered by a computation with $NbExp_{\max}$ decomposition relative to one without $NbExp_{\max}$ decomposition. Tests performed on IBM Power4, using the *block_blas* routine (with optimal $s_{\text{block}}$) and the scheme with *recopies* for an octree height equal to 5 ($NbExp = 2744$) and 6 ($NbExp = 27{,}000$). The corresponding values for $s_{\text{block}}$ and $NbExp_{\max}$ are also given.

decomposition. But for higher values of $P$, the best $s_{\text{block}}$ values were lower than $P$ which justifies the recursive decomposition: see Fig. 8.

Finally, with these optimal $s_{\text{block}}$ values, the relevance of the $NbExp_{\max}$ decomposition is shown on Fig. 8 for the scheme with *recopies*. With growing octree heights, the numbers of expansions *NbExp* treated with one single matrix–matrix product increases, and the gain offered by the $NbExp_{\max}$ decomposition increases thus too. With *row* and *slice* storages, $NbExp_{\max}$ decomposition will be likewise relevant when the *NbExp* value exceeds the optimal $NbExp_{\max}$ value.

### 3.3.2. Recopies and special data storages

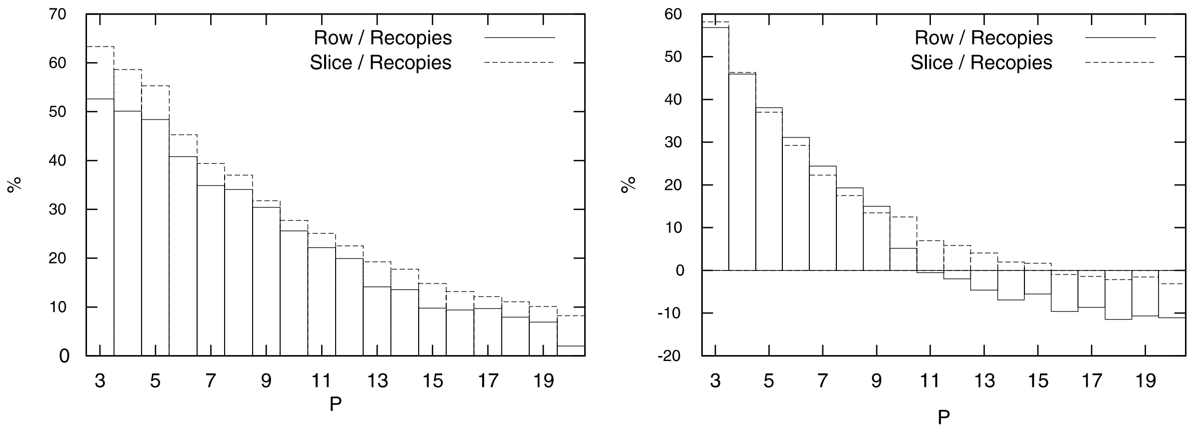
In order to compare the scheme with *recopies* with *row* and *slice* storages, CPU times have been measured on the full downward pass of the FMM. As mentioned in [30], the cost of copying vectors relatively to the cost of the matrix–matrix product decreases for growing values of $P$: copies of multipoles and local expansions are indeed performed in $\mathcal{O}(NbExp \times P^2)$ while the matrix–matrix product requires $\mathcal{O}(NbExp \times P^4)$ operations. This is more obvious with double height kernel than with single height kernel since the amount of computation for the matrix–matrix product is much more costly with double height, while the cost of *recopies* is the same. Fig. 9 shows the gain of *row* and *slice* storages relative to the scheme with *recopies* according to different values of $P$ for an octree height of 6: *row* and *slice* storages offer thus better gains relative to *recopies* for low values of $P$, and these gains are always higher for single height kernel.

These gains are also influenced by the height of the octree: with growing heights of the octree, the number of expansions treated with one matrix–matrix product increases for *row* and *slice* storages (see Table 1) and the proportion of *blank boxes* decreases for *slice* storage. And for low heights of the octree, the use of *recopies* may sometimes be faster as in Fig. 9.

*Slice* storage is here more efficient than *row* storage because of the too small *NbExp* for *row* storage at this height: for greater heights, *row* storage is a little bit faster. Moreover *slice* storage needs very long consecutive memory areas that cannot be allocated for too high values of $P$ or too high heights when memory allocation with *row* storage may succeed. For these reasons we prefer *row* storage, and since the *NbExp* values enabled by *slice* storage seem to be high enough, *level* storage has also been discarded.

At last, as mentioned in Sections 2.3 and 2.4, no BLAS routine can be applied to the FFT and rotation improvements, and attempts to apply the scheme with *recopies* or *row* data storage have failed in improving the performances of these computations.

Now that we have determined the best BLAS implementations, namely *block_blas* and *full_blas* (depending on the *M2L* kernel height), with *row* data storage, we are able to practically compare them with the other *M2L* improvements.



(a) *block_blas* routine (single height kernel). (b) *full_blas* routine (double height kernel).

Fig. 9. Gain in percentage offered by *row* and *slice* storages relative to the scheme with *recopies* for downward pass CPU times with an octree of height 6. Tests performed on IBM Power4.

## 4. Comparison of the *M2L* improvements

Thanks to the implementation of the schemes based on FFT, rotations and BLAS routines in our FMB code, we can now precisely compare all these improvements of the *M2L* operator. The comparison with the plane wave scheme will be done afterwards thanks to the FMMPART3D code.

### 4.1. Memory requirements

We first compare the memory requirements of the different schemes since it may be the first choice criterion. We here focus only on the memory used for the expansions: the memory used for the particles remains indeed unchanged when computing *M2L* operator differently. In a nutshell, *M2L* computation with rotations has very low extra memory needs (see Section 2.4). BLAS computation needs multipole vectors, whose size is roughly twice the size of the multipole expansions, and extra memory for the *M2L* transfer matrices, especially with double height kernel since these matrices are dense in this case. *slice* data storage requires also extra memory for the *blank boxes*. At last *M2L* computation with block or non-block FFT uses bigger arrays for its multipole expansions and its *M2L* transfer functions: these arrays are roughly four times bigger with single height *M2L* kernel and 16 times bigger with double height kernel. Besides, the polynomial scaling does not affect the memory requirements of the non-block FFT computation. Using more detailed theoretical estimations that can be found in [22], we have plotted in Fig. 10, for each scheme, the ratio of its memory need to the classic *M2L* memory need, using an octree of height 6 and only multiples of the FFT block size.

As predicted, while the rotation requirements are almost unnoticeable, and the BLAS ones remain moderate, FFT extra memory appear as problematic, especially for double height *M2L* kernel. The same ratios apply for other FFT block sizes, and the memory requirements of the non-block FFT computation are the same for these multiples of the FFT block size (and similar otherwise). Moreover the ratio of 2 for BLAS in the double height kernel with high values of $P$ is due to the dense *M2L* transfer matrices: for greater octree heights, this ratio remains close to 1.5 since the number of *M2L* functions is constant while the number of cells in the octree grows exponentially. At last the additional cost of the *blank boxes* in *slice* storage over *row* storage is very small.

### 4.2. CPU times for single and double heights M2L kernel

Fig. 11(a) compares the different schemes with single height *M2L* kernel. If the BLAS version (here with *row* storage) outperforms the version with the classic *M2L* and the one with rotations, the block FFT and the FFT with polynomial scaling are faster. However in this test, the FFT with block size 4 is unstable for

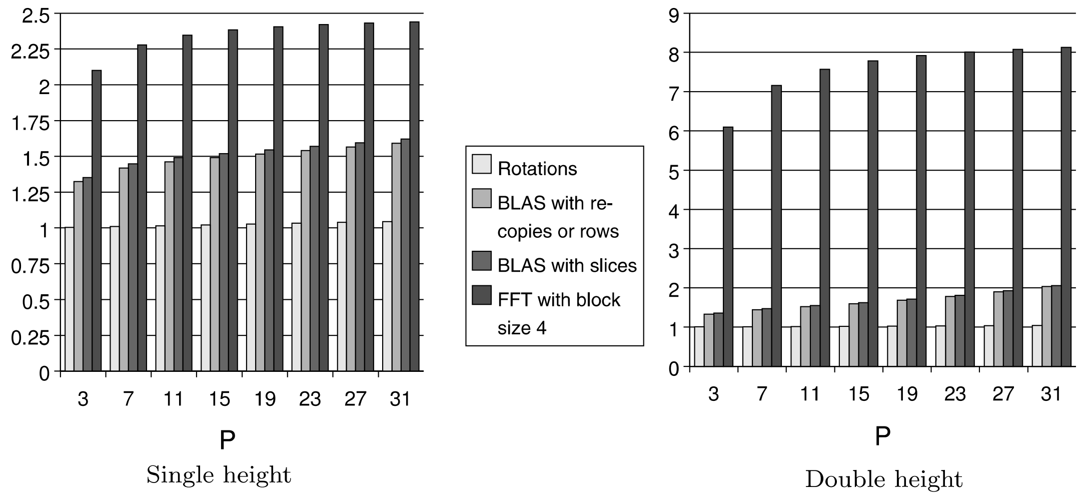


Fig. 10. Memory requirements of the different *M2L* computation schemes for an octree height of 6 (ratios to classic *M2L* computation requirements).

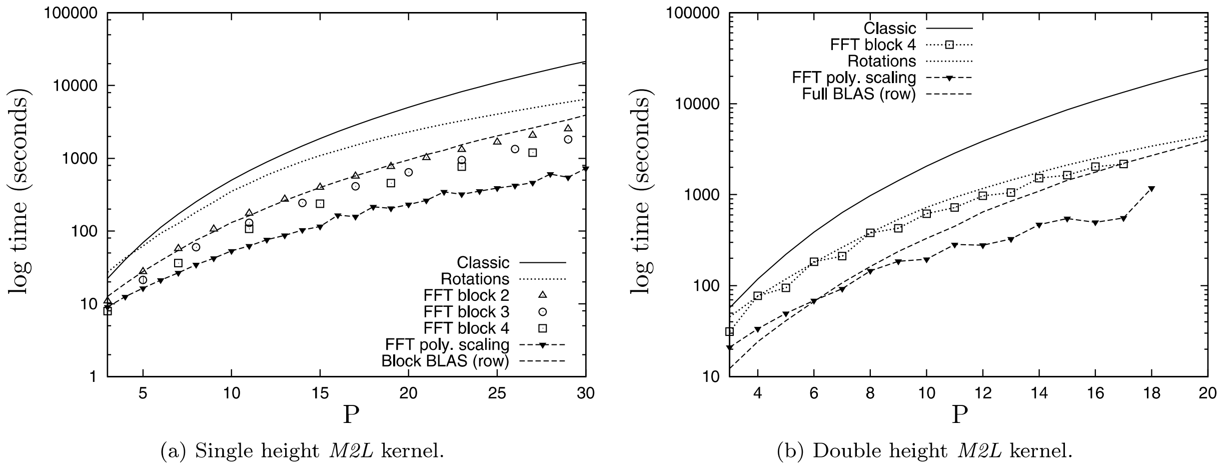


Fig. 11. Logarithmic downward pass CPU times of the different *M2L* computation schemes for 1 million particles with a fixed octree height of 5. Tests are performed on an IBM Power3 with 2 GB of memory.

$P \geqslant 14$, and the one with block size 3 for $P \geqslant 23$. Moreover, as we will see in Section 4.4, the BLAS computation with *row* or *slice* storages is more efficient with greater octree heights.

With double height kernel, as shown in Fig. 11(b), the use of BLAS clearly outperforms all $\mathcal{O}(P^3)$ computation schemes: only the $\mathcal{O}(P^2 \log P)$ FFT scheme with polynomial scaling is faster, but unstable for $P > 17$. For the BLAS, we have plotted only the *row* data storage, but *slice* storage and the scheme with *recopies* have similar performances. For the FFT, the block version with blocks of size 4 have been plotted; in this test, it is only stable for $P \leqslant 8$, and more stable FFT with lower block sizes are slower. Moreover, on the IBM Power3 the 2 GB of memory were insufficient for FFT of block size 4 with $P > 17$, and for FFT with polynomial scaling with $P > 18$. Finally, as for single height kernel, the BLAS computations with *row* and *slice* storages become even more efficient for greater heights of the octree (see Section 4.4).

## 4.3. Computational efficiency

In order to illustrate the efficiency of the BLAS versions that makes them faster than the $\mathcal{O}(P^3)$ schemes, we present the Table 2 that shows the millions of floating point operations per seconds (MFLOPS) when com-

Table 2
Computational efficiencies of the different *M2L* computation schemes

| | Single height *M2L* kernel | | Double height *M2L* kernel | |
|---|---|---|---|---|
| | $P = 7$ | $P = 15$ | $P = 7$ | $P = 15$ |
| Classic (%) | 9.4 | 16.3 | 14.5 | 18.3 |
| Rotations (%) | 5.4 | 7.8 | 8.4 | 10.9 |
| FFT with block size 4 (%) | 4.5 | 13.0 | 12.4 | 18.4 |
| Level 3 BLAS (%) | 46.4 | 74.0 | 85.9 | 89.2 |

puting *M2L*. The results are shown as percentages of the peak performance of the IBM Power3 used: 1500 MFLOPS. They have been computed with the Hardware Performance Monitor (HPM) Toolkit (version 2.4.3) as the average MFLOPS rate over all *M2L* computations of a full FMM computation. For level 3 BLAS, we have used *block_blas* and *full_blas* routines for respectively single and double height kernels, with an octree of height 5 and with *recopies* (the additional cost of copying the expansions is not considered here since we focus on the BLAS routine efficiency). It takes into account all cells, i.e. with and without complete interaction list. The height of the octree does not matter for classic *M2L*, rotations and FFT. These results can however not be directly compared from one row to the other: we recall here that the operations count differs! However it clearly illustrates how efficiently the processor is used in the different implementations, and why BLAS computations outperform in practice computation schemes with lower theoretical operation counts.

This table also indicates that our decomposition of the matrix–matrix product for single height kernel in the *block_blas* routine, though satisfactory, is not optimal when compared to the *full_blas* routine efficiency of the double height kernel, especially for low values of $P$.

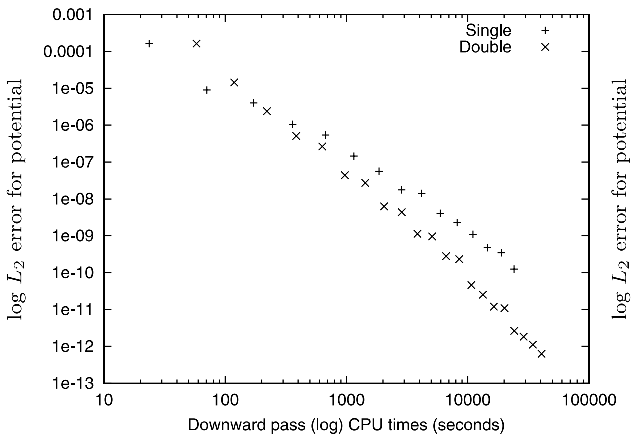
## 4.4. Single versus double height kernels

We have already seen (see Theorem 4) that a sharp error bound has been theoretically proven only for the single *M2L* height kernel. The double height *M2L* kernel is certainly more precise, and for a given precision, it would therefore require a lower value for $P$ than the single height one. But since no precise error bound is available for double height kernel, we cannot a priori know what this lower $P$ will be.

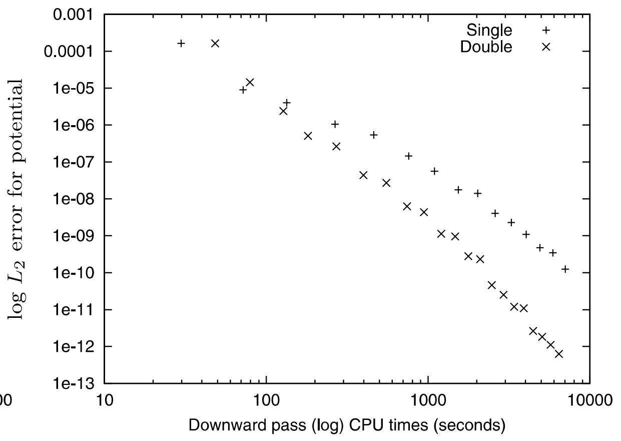
That is why we compare here CPU times with practical accuracies for both kernel heights. As already exposed (see for example [20] for double height *M2L* kernel, and [17] for single height kernel), these practical accuracies are better than the theoretical ones which correspond to worst-case errors. These worst-case errors are indeed obtained with spherical regions, while we use in fact smaller cubical cells because of the octree. Like in the articles [11,12], we choose to study the $L_2$ error defined in Section 2.3.1. Contrary to an absolute error, this $L_2$ error enables indeed problem independent values, and contrary to a maximum error over all particles, it smoothes the possible discontinuities that appear in the potential error for particles crossing cell boundaries (see [17]). Like in Section 2.3.1, we use here 1000 randomly chosen bodies as references for the exact potential in the $L_2$ error computation.

For our tests, we consider a gravitational potential computed for an uniform distribution with 1 million bodies and an octree of height 5, which results in 31 bodies per leaf in the mean. We recall that when the number of bodies per leaf increases, the part of the near field (directly computed) in the potential becomes higher and the potential becomes thus more precise. Fig. 12 presents, for each *M2L* scheme, the tradeoff between practical error and CPU times (downward pass only) with both single and double height kernels. The scales are logarithmic, and the values of $P$ plotted for single height kernel range from 3 to 31 with step 2, while for double height kernel they range from 3 to 23 with a step of 1 (except for cases where the 2 GB memory are insufficient). $P = 23$ with double height kernel corresponds indeed to the first accuracy below $1.0 \times 10^{-12}$.

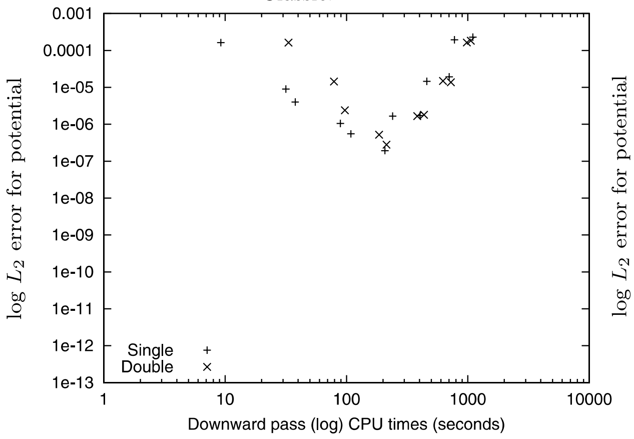
As far as classic *M2L* is concerned, the single height kernel is more efficient for low precisions and the double height one for high precisions. And as theoretically justified in Sections 2.3, 2.4 and 4.3, the rotation and BLAS computations generally favor the double height kernel, except for the low values of $P$ where the single height kernel is more efficient. On the other hand, the FFT improvement is generally more efficient with single height kernel, especially with the block version. Moreover, the numerical instabilities of the block FFT scheme with block size 4 prevent from reaching a precision below $10^{-7}$ in these tests. For the non-block FFT with



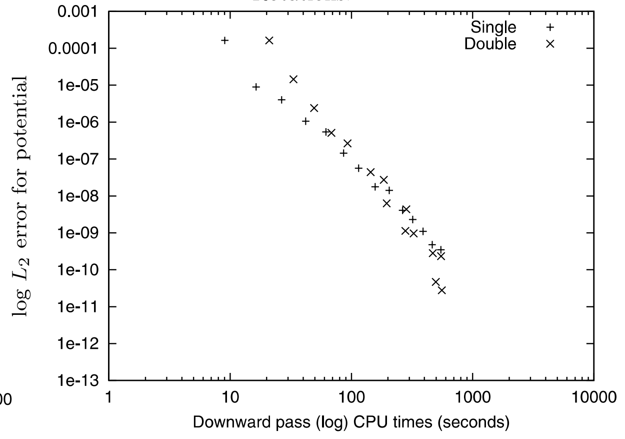
Classic.



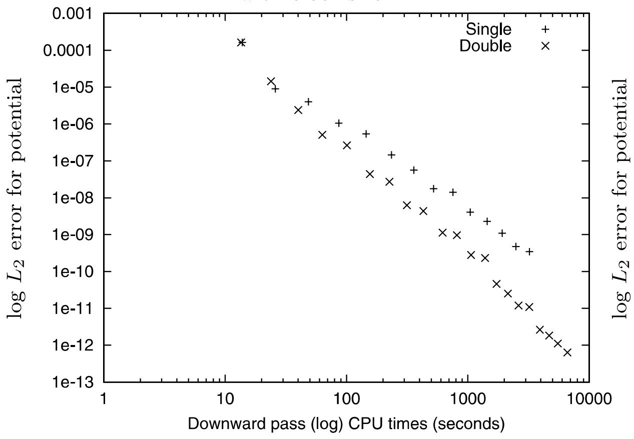
Rotations.



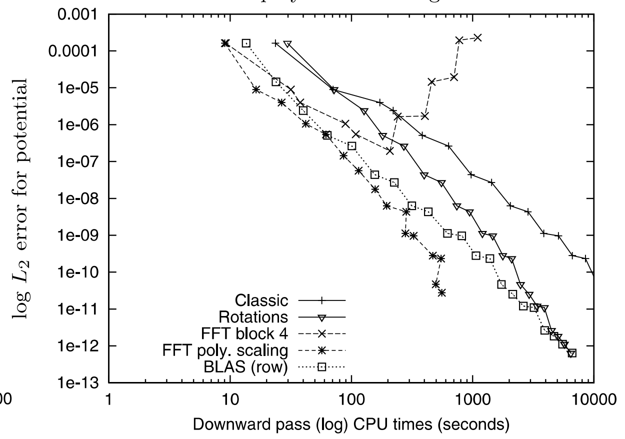
FFT with block size 4.



FFT with polynomial scaling.



BLAS (*block blas / full blas*, *row* storage).



Best kernel height for each accuracy.

Fig. 12. Tradeoffs between practical accuracies and CPU times with single and double height kernels, for an uniform distribution with 1 million bodies and an octree height fixed to 5. Tests run on IBM Power3 with 2 GB memory.

polynomial scaling, the memory requirements prevent from running tests with $P > 17$ and double height kernel: this is anyway the numerical stability limit of this computation scheme (see Section 2.3.2).

Therefore we still have to compare the best of each scheme: this is done in Fig. 12 where we select the best kernel height for each accuracy and present all the schemes on the same plot. The BLAS version is then clearly more efficient than all $\mathcal{O}(P^3)$ schemes. By examples, for a practical error below $1.0 \times 10^{-6}$ the gain of the BLAS over the FFT (respectively the rotations) is 42% (respectively 65%). One can also see that for the greatest $P$ values, the rotation scheme becomes as fast as the BLAS version: this is of course the aftermath of the lower operation count for the rotation scheme. This comparison thus fully validates the relevance of our new BLAS version compared to these $\mathcal{O}(P^3)$ *M2L* improvements.

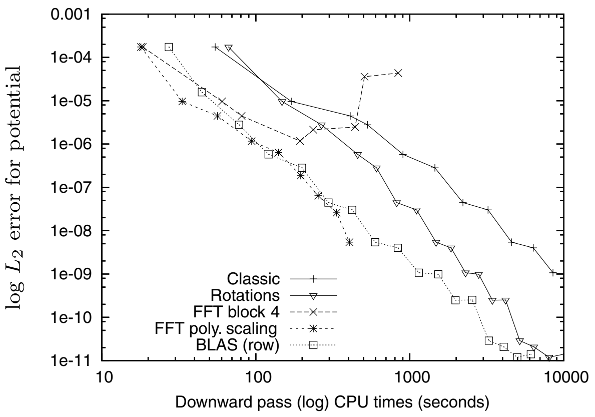
The $\mathcal{O}(P^2 \log P)$ FFT with polynomial scaling appear however always faster than our BLAS version, but its limitations concerning memory requirements and numerical stability are also clear. The memory requirements are even more problematic with greater octree height as shown in Fig. 13(a) and (b). Fig. 13(a) also shows that for greater octree height the BLAS computation with *row* (or *slice*) storage becomes more efficient. Indeed, with growing octree heights the length of rows and slices increases at the leaf level which represents most of the runtime: the number of expansions treated in one level 3 BLAS call thus increases, as well as the efficiency of the BLAS routines. In Fig. 13(a), the BLAS computation scheme is then clearly competitive with the FFT with polynomial scaling.

Finally, we have also run tests on a Linux PC with Intel Pentium 4 processor, using the ATLAS BLAS library (version 3.6.0) [36,37], in order to show how our results rely on the architecture used. In Fig. 13(b), one can see that the BLAS version losts here some efficiency compared to the other computation schemes, mainly because the ATLAS routines on Intel Pentium processors are less efficient than vendor IBM BLAS implementation on IBM processors. Nevertheless, our BLAS version is still faster than the classic and rotation schemes, while the FFT schemes have too high memory requirements. This therefore validates our BLAS version on other architectures.

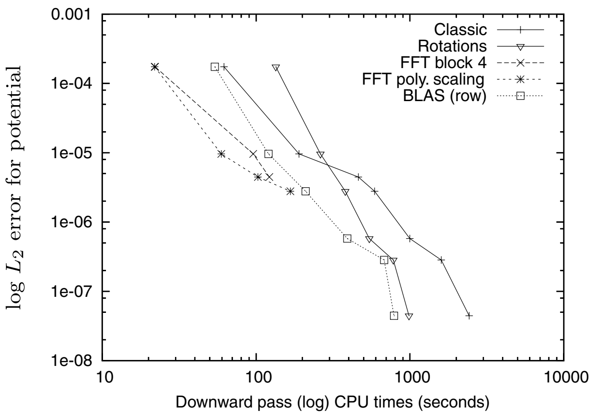
### 4.5. Comparison with plane waves

In order to compare our BLAS version with the scheme based on plane wave expansions, we will study the CPU times of both our FMB code and the FMMPART3D code (version 1.0) according to the practical accuracies. This comparison has been done on the Linux PC with Intel Pentium 4 processor presented in Section 4.4. Our FMB code is compiled with GCC 4.0.4 and uses the ATLAS BLAS library. The FMMPART3D code is available as a precompiled library (Lahey/Fujitsu Fortran 95 compiler, version 6.2).

We start by outlining that the articles [11,12] do not introduce the *Outer* and *Inner* functions (see Section 2.1): their expressions of *M2M*, *M2L* and *L2L* operators are thus more complicated, and therefore more expensive than ours in terms of operation count.



(a) On IBM Power5 (1.9 Ghz, L2 cache size: 1.9 MB, L3 cache size: 36MB) with 6GB of available memory (located at M3PEC center in University of Bordeaux 1).



(b) On Intel Pentium 4 (2.6 Ghz, L1 cache size: 8 KB, L2 cache size: 512 KB) with 2GB of memory.

Fig. 13. Tradeoffs between practical accuracies and CPU times with single and double height kernels, for an uniform distribution and an octree height fixed to 6.

As for previous comparisons, we only consider here uniform distributions of particles. Like in FMM-PART3D, we use as practical error measurement the $L_2$ norm defined in Section 2.3.1, with direct computation used to compute the exact potential of 1000 randomly chosen bodies.

The FMMPART3D code offers the four following theoretical precisions ensured by the plane wave scheme: $1.6 \times 10^{-3}$ (*low*), $1.3 \times 10^{-6}$ (*medium*), $1.1 \times 10^{-9}$ (*high*) and $1.0 \times 10^{-12}$ (*very high*). The corresponding $P$ values are presented in Table 3. As usual with the FMM (see Section 4.4), we notice in the following tests that the practical errors obtained with FMMPART3D are much lower than these theoretical error bounds.

For FMB, the values of $P$ range from 1 to 28, with step 1, so that we cover the whole range of precisions available in FMMPART3D. The FMB code selects the best computation scheme with level 3 BLAS routines depending on the octree level: the *row* data storage mode when the level $l$ in the octree is high enough ($l \geqslant 5$), and otherwise the scheme with *recopies* (see Section 3.3.2).

Moreover, some algorithmic differences between the two codes have to be precised.

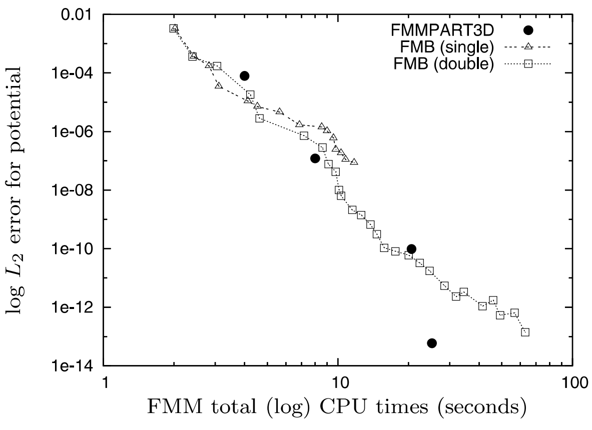
– With FMB, we use single height and double height *M2L* kernels. The single height kernel is however presented here only up to $P = 15$ since, as noticed in Section 4.4, the single height kernel is more efficient than the double height one only for the low values of $P$. In FMMPART3D and in the articles [11,12], the choice of the kernel height is unprecised, which suggests a double height kernel (more casual and more common).
– As far as the octree height is concerned, this one is optimally set by the user in FMB in order to balance the near field and the far field computations. With growing values of $P$, the cost of the far field computation increases, and we may have to reduce the octree height. On the other hand, FMMPART3D is an adaptive FMM code, whose algorithm is presented in [12,38]. This algorithm imposes a maximal number of particles per leaf in the octree: this maximum value can not be tuned by the user in FMMPART3D.
– In FMMPART3D, the operations *M2M* and *L2L* are computed thanks to a scheme with rotations in $\mathcal{O}(P^3)$ (see Section 2.4), whereas in FMB we use the classic computation in $\mathcal{O}(P^4)$. Nevertheless these operations are clearly minority in the total computation cost of the FMM for uniform distributions.

Figs. 14–16 present the tradeoff between CPU times and $\varepsilon_{L_2}$ error for both codes, and for uniform distributions of 100,000, 1 million, and 3 million particles. In each figure, the four dots for FMMPART3D correspond to the four available precisions. Several conclusions can be drawn from these figures.

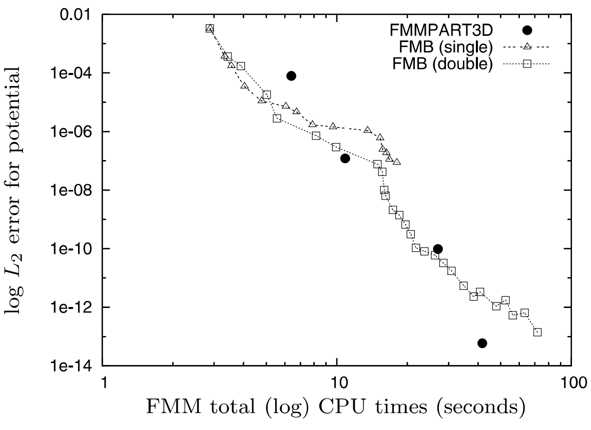
– We first notice with Table 3 that for each precision available in FMMPART3D, the $P$ value required to reach (or surpass) the corresponding *practical* precision (that is to say the error obtained in practice) with FMB is much lower than the $P$ value used in FMMPART3D. Such a gap is probably due to a loss of the FMM accuracy because of the plane wave introduction. The error generated in practice with the plane wave expansions may be greater than the one due to the spherical harmonic expansions, even though the practical error of FMMPART3D remains of course below the theoretical error bounds. A decrease of this additional error due to the plane wave scheme could only be done at the expense of an increase in its operation count.
– We can also remark that FMB is more efficient, with respect to FMMPART3D, when both the potential and the force are computed (in comparison with the computation of the potential only). This can be explained by a better implementation of our direct computation for the force, or by a more efficient computation of the evaluation of the force from the local expansions. As this is not directly related to the *M2L* computation, we focus on the figures where only the potential is computed.

Table 3
For each of the four precisions available in FMMPART3D: $P$ values used in FMMPART3D and $P$ values in FMB that enable to reach (or surpass) the corresponding *practical* precision obtained with FMMPART3D

| $P$ | Low | Medium | High | Very high |
|---|---|---|---|---|
| FMMPART3D | 10 | 19 | 29 | 40 |
| FMB (double height *M2L* kernel) | 4 | 8 | 16 | 26–28 |
| FMB (single height *M2L* kernel) | 4 | 14 | – | – |

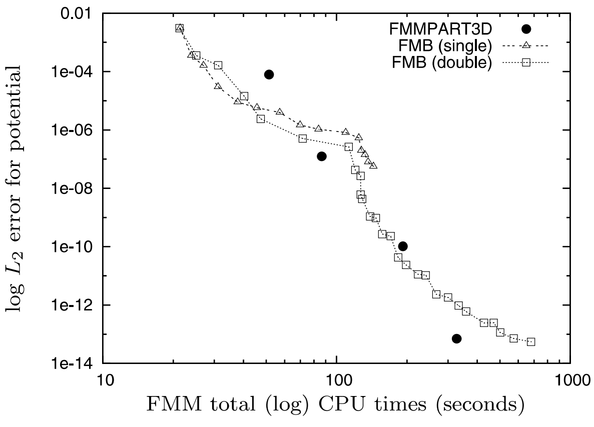


(a) Only potential computation.

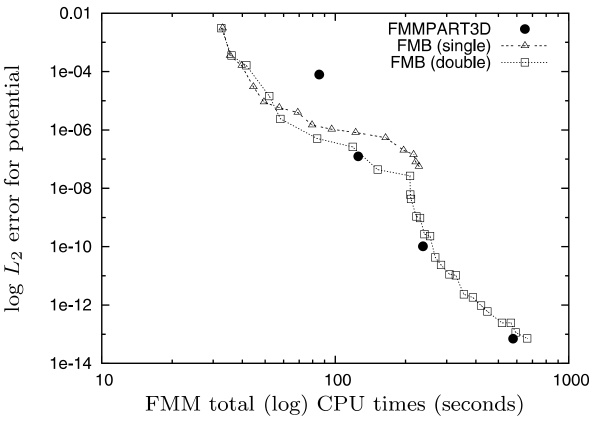


(b) Force and potential computed.

Fig. 14. Comparison between FMMPART3D and FMB for an uniform distribution of 100,000 particles.

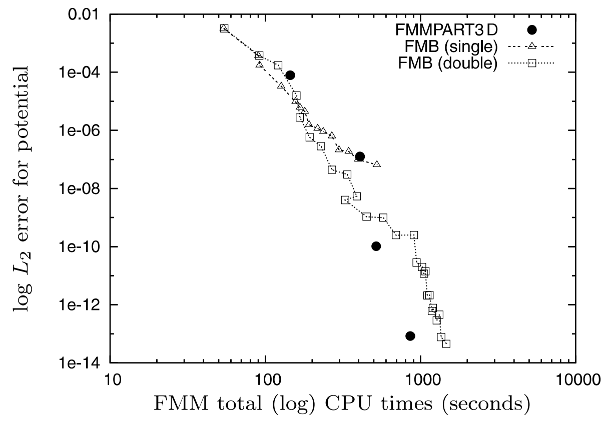


(a) Only potential computation.

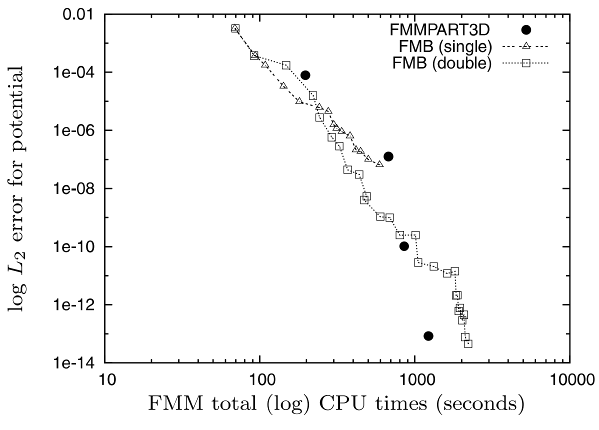


(b) Force and potential computed.

Fig. 15. Comparison between FMMPART3D and FMB for an uniform distribution of 1 million particles.



(a) Only potential computation.



(b) Force and potential computed.

Fig. 16. Comparison between FMMPART3D and FMB for an uniform distribution of 3 million particles.

– As announced, the single height *M2L* kernel is more efficient than the double height kernel for the low values of $P$ (concretely for $2 \leqslant P \leqslant 5$ or 6).
– Finally, if we compare the CPU times for a given precision and for the computation of the potential only, FMB is always better for the first precision (*low*), with for example a gain of 39% with $P = 4$ (single height *M2L* kernel) in the case of 1 million particles. For the two following precisions (*medium* and *high*), FMB is as fast as FMMPART3D, but slower for the highest precision (*very high*).

This perfectly illustrates the gain in performance of the BLAS computation scheme which keeps the operation count of the classic *M2L* computation but greatly reduces the underlying constant factor: for low and medium values of $P$ the best gain is offered by the BLAS routines, and the gain in operation count offered by the plane wave scheme is crucial only for the highest values of $P$.

The breaks in the FMB plots correspond to decrements of the octree height. The octree heights being lower in the case of 100,000 particles, our computation scheme with BLAS is less efficient, but the same behavior is observed. For the cases with 1 million and 3 million particles, the octree heights are the same (with however different decrements) and the differences in the plots are essentially due to the more important part of the direct computation with 3 million particles. Due to compiler and licence restrictions we could not run FMMPART3D on other architectures with more memory (like SMP nodes[11]), but in this case we would have been able to run bigger test cases with greater octree height, where our BLAS scheme would be likely to be even more efficient (see Section 3.3.2).

As exposed in Section 1 of this paper, the common precisions for Laplace equation are lower or equal to roughly $10^{-7}$. This corresponds to the range of $P$ values where our FMB code is either faster than, or as fast as, the FMMPART3D code. By the way, this may explain why the plane wave scheme has not been used with Laplace equation since the article [12]: contrary to Maxwell and Helmholtz equations, where the $P$ values are high, the values of $P$ required in Laplace equation limit the gain offered by a the operation count in $\mathcal{O}(P^2) + \mathcal{O}(P^3)$, and the underlying constant factor of this operation count is not negligible in this case.

## 5. Conclusion

In this paper, we have presented an overall study of the most efficient implementation of the fast multipole method for the serial computation of gravitational or electrostatic simulations of uniform distributions.

A detailed study of the error bound has lead us to two expressions of the *M2L* operator that converts a multipole expansion into a local expansion: while the *double height M2L kernel* is generally more efficient for medium and high precisions, the *single height M2L kernel* is faster for low precisions and is the only one to ensure a sharp error bound. For each *M2L* expression, we have efficiently implemented the *block FFT*, the *FFT with polynomial scaling* and the *rotation* improvements. To our knowledge, this is the first implementation of the FFT enhancement for double height *M2L* kernel, and it has been shown that numerical instabilities remain even with the block FFT. Moreover, we have given the best parameters for the polynomial scaling of the 3*D* FMM, and we have determined its practical limits concerning numerical stability.

As an alternative, a BLAS (Basic Linear Algebra Subprograms) version has been proposed for the dense matrices of the double height kernel as well as for the sparse matrices of the single height one. A scheme with *recopies* has first made possible the use of level 3 BLAS resulting in impressive speedups. Special data storages for the expansions either by *rows* or by *slices* have then enabled us to avoid the additional cost of *recopies*, especially for low precisions.

Memory requirement estimations and CPU times from practical simulations have been used to precisely draw the first comparison of all these different schemes. When comparing out $\mathcal{O}(P^4)$ BLAS version with $\mathcal{O}(P^3)$ methods, it appears that the BLAS version and the FFT improvement with blocks are the most efficient ones. While the BLAS version is always faster in case of double height kernel, the block FFT is faster with single height kernel for high precisions. However the memory requirements of the block FFT as well as the

[11] On SMP nodes, vendor BLAS implementations are also likely to be faster than the ATLAS routines.

remaining numerical instabilities, precisely for high precisions, limit severely the benefit it offers for runtime in this case. When comparing the tradeoff between practical accuracy and runtime for both *M2L* kernel heights, the BLAS version is then the most efficient, while introducing no numerical instabilities and low extra memory requirements.

The $\mathcal{O}(P^2 \log P)$ FFT with polynomial scaling is however generally faster than our BLAS version. But its memory requirements limits also severely its usage, and this computation scheme is not reliable because of its remaining numerical instabilities. Moreover, for growing octree heights our BLAS version becomes more efficient, and hence as fast as the FFT with polynomial scaling.

Finally, we have also compared our BLAS computation scheme with the plane wave scheme thanks to the FMMPART3D code: in the range of common precisions for Laplace equation in astrophysics and molecular dynamics, the gain offered by the BLAS routines is either greater than, or equal to, the one obtained with the plane wave scheme.

We have already extended this BLAS scheme to the adaptive version of the FMM [39], and we are currently parallelizing it on shared and distributed memory architectures. In the near future, our BLAS implementation may appear even more appealing in the context of heterogeneous high performance computing with special-purpose hardware, such as Graphics Processing Units (GPU) or Cell processors. Indeed, these hardwares have (or will have) a BLAS library whose integration in our FMB code will be straightforward. Finally, this BLAS approach could also be extended to other potentials whose FMM operators can be written as matrix products.

**Acknowledgments**

We are grateful to Guillaume Latu, ICPS-LSIIT (Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection) and INRIA CALVI project, for several helpful discussions on the underlying techniques of BLAS and their integration in the FMM.

## Appendix A. Proof of the Theorem 4

**Proof.** With the notations and assumptions of Section 2.2, we consider a multipole expansion, centered in $\mathbf{X_1}$, and a local expansion, centered in $\mathbf{X}_2$, to express the potential at $\mathbf{Z}$ due to the particle in $\mathbf{Q}$. Since the condition $R > r_1 + r_2$, which guarantees the convergence of both multipole and local expansions, implies $\|(\mathbf{Q} - \mathbf{X_1}) - (\mathbf{Z} - \mathbf{X_2})\| < \|\mathbf{X_2} - \mathbf{X_1}\|$, and since $I_n^l(-\mathbf{X}) = (-1)^n I_n^l(\mathbf{X})$ the potential in $\mathbf{Z}$ writes, with the classical translation Theorem (Theorem 1):

$$\Phi(\mathbf{Z}) = \sum_{n=0}^{+\infty} \sum_{l=-n}^{n} I_n^{-l}((\mathbf{Z} - \mathbf{X_2}) - (\mathbf{Q} - \mathbf{X_1}))O_n^l(\mathbf{X_2} - \mathbf{X_1}).$$

Thanks to the Inner-to-Inner translation Theorem (Theorem 3), we obtain

$$\Phi(\mathbf{Z}) = \sum_{n=0}^{+\infty} \sum_{l=-n}^{n} \sum_{j=0}^{n} \sum_{k=-j}^{j} (-1)^j I_j^k(\mathbf{Q} - \mathbf{X_1}) I_{n-j}^{-l-k}(\mathbf{Z} - \mathbf{X_2}) O_n^l(\mathbf{X_2} - \mathbf{X_1}). \tag{A.1}$$

We emphasize here that Eq. (A.1) is just a rewriting of Eq. (4): that is why when truncating the series in Eq. (A.1) for $n > P$, we obtain the same error bound as in Proposition 2. Thus we have

$$\Phi_P(\mathbf{Z}) = \sum_{n=0}^{P} \sum_{l=-n}^{n} \sum_{j=0}^{n} \sum_{k=-j}^{j} (-1)^j I_j^k(\mathbf{Q} - \mathbf{X_1}) I_{n-j}^{-l-k}(\mathbf{Z} - \mathbf{X_2}) O_n^l(\mathbf{X_2} - \mathbf{X_1}).$$

After an inversion of the two finite summations on the degrees $n$ and $j$ and reindexing $n - j \to n'$ and $l + k \to l'$ we obtain

$$\Phi_P(\mathbf{Z}) = \sum_{j=0}^{P} \sum_{k=-j}^{j} \sum_{n'=0}^{P-j} \sum_{l'=-(n'+j)+k}^{n'+j+k} (-1)^j I_j^k(\mathbf{Q} - \mathbf{X_1}) I_{n'}^{-l'}(\mathbf{Z} - \mathbf{X_2}) O_{n'+j}^{l'-k}(\mathbf{X_2} - \mathbf{X_1}).$$

We remember here that $I_{n'}^{-l'}(\mathbf{x})$ imposes $-n' \leqslant l' \leqslant n'$. In the same way, $I_j^k(\mathbf{x})$ imposes $|k| \leqslant j$, that is to say $j+k \geqslant 0$, and $k-j \leqslant 0$. Moreover,

$$\begin{aligned}&j+k \geqslant 0 \Rightarrow n'+j+k \geqslant n',\\&k-j \leqslant 0 \Rightarrow -(n'+j)+k = -n'+(k-j) \leqslant -n'.\end{aligned}$$

Under these conditions and after reindexing $-l' \to l$ and $n' \to n$ and another inversion of summations on the degrees, we obtain

$$\Phi_P(\mathbf{Z}) = \sum_{n=0}^{P}\sum_{l=-n}^{n}\left(\sum_{j=0}^{P-n}\sum_{k=-j}^{j}(-1)^j I_j^k(\mathbf{Q}-\mathbf{X_1})O_{n+j}^{-l-k}(\mathbf{X_2}-\mathbf{X_1})\right)I_n^l(\mathbf{Z}-\mathbf{X_2}).$$

For all $\mathbf{Q}$ in $\mathcal{B}_1$, the multipole expansion terms $M_j^k\ \forall(j,k) \in \mathbb{N}\times\mathbb{Z}$ with $|k| \leqslant j$, being defined by $M_j^k = (-1)^j I_j^k(\mathbf{Q}-\mathbf{X_1})$, we have thus proved that

$$\Phi_P(\mathbf{Z}) = \sum_{n=0}^{P}\sum_{l=-n}^{n} L_n^l I_n^l(\mathbf{Z}-\mathbf{X_2}),$$

where the $L_n^l$ denote the local expansion terms, centered in $\mathbf{X}_2$, due to the unit charge located in $\mathbf{Q}$, and obtained thanks to the $M2L$ operator with single height $M2L$ kernel applied to the multipole expansion terms $M_j^k$ as

$$L_n^l = \sum_{j=0}^{P-n}\sum_{k=-j}^{j} M_j^k O_{n+j}^{-l-k}(\mathbf{X_2}-\mathbf{X_1}). \qquad \square$$

## References

[1] L. Greengard, V. Rokhlin, A fast algorithm for particle simulations, Journal of Computational Physics 73 (1987) 325–348.

[2] L. Greengard, The Rapid Evaluation of Potential Fields in Particle Systems, MIT Press, Cambridge, MA, 1988.

[3] W. Elliott, Multipole algorithms for molecular dynamics simulation on high performance computers, Tech. Rep. 95-003, Duke University, Department of Electrical Engineering, doctoral dissertation, 1995.

[4] E. Darve, Méthodes multipôles rapides: résolution des équations de Maxwell par formulations intégrales, Ph.D. thesis, Université Paris 6, 1999.

[5] K. Nabors, J. White, Fastcap: a multipole accelerated 3-D capacitance extraction program, IEEE Transactions on Computer-Aided Design 10 (11) (1991) 1447–1459.

[6] J.P. Singh, C. Holt, T. Totsuka, A. Gupta, J. Hennessy, Load balancing and data locality in adaptive hierarchical *N*-body methods: Barnes-Hut, fast multipole, and radiosity, Journal of Parallel and Distributed Computing 27 (1995) 118–141.

[7] J.C. Carr, R.K. Beatson, J. Cherrie, T.J. Mitchell, W.R. Fright, B.C. McCallum, T.R. Evans, Reconstruction and representation of 3D objects with radial basis functions, in: ACM SIGGRAPH 2001, Los Angeles, CA, 2001, pp. 67–76.

[8] W.D. Elliott, J.A. Board Jr., Fast Fourier transform accelerated fast multipole algorithm, SIAM Journal on Scientific Computing 17 (2) (1996) 398–415.

[9] L. Greengard, V. Rokhlin, On the efficient implementation of the fast multipole algorithm, Tech. Rep. RR-602, Department of Computer Science, Yale University, New Haven, CT, 1988.

[10] C.A. White, M. Head-Gordon, Rotating around the quartic angular momentum barrier in fast multipole method calculations, Journal of Chemical Physics 105 (12) (1996) 5061–5067.

[11] L. Greengard, V. Rokhlin, A new version of the fast multipole method for the Laplace equation in three dimensions, Acta Numerica 6 (1997) 229–269.

[12] H. Cheng, L. Greengard, V. Rokhlin, A fast adaptive multipole algorithm in three dimensions, Journal of Computational Physics 155 (1999) 468–498.

[13] E. Darve, P. Havé, Efficient fast multipole method for low-frequency scattering, Journal of Computational Physics 197 (1) (2004) 341–363.

[14] J. Kurzaka, B.M. Pettitt, Communications overlapping in fast multipole particle dynamics methods, Journal of Computational Physics 203 (2) (2005) 731–743.

[15] N.A. Gumerov, R. Duraiswami, Recursions for the computation of multipole translation and rotation coefficients for the 3-D Helmholtz equation, SIAM Journal on Scientific Computing 25 (4) (2003) 1344–1381.

[16] J. Dongarra, J.D. Croz, S. Hammarling, I. Duff, A set of level 3 basic linear algebra subprograms, ACM Transactions on Mathematical Software 16 (1) (1990) 1–17.

[17] W. Rankin, Efficient Parallel Implementations of Multipole based *N*-body algorithms, Ph.D. Dissertation, Duke University, Department of Electrical Engineering, April 1999.
[18] R. Capuzzo-Dolcetta, P. Miocchi, A comparison between the fast multipole algorithm and the tree-code to evaluate gravitational forces in 3-D, Journal of Computational Physics 143 (1) (1998) 29.
[19] D. Soelvason, H. Petersen, Error estimates for the fast multipole method, Journal of Statistical Physics 86 (1997) 391–420.
[20] C.A. White, M. Head-Gordon, Derivation and efficient implementation of the fast multipole method, Journal of Chemical Physics 101 (8) (1994) 6593–6605.
[21] H. Petersen, E. Smith, D. Soelvason, Error estimates for the fast multipole method. II. The three-dimensional case, Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences 448 (1995) 401–418.
[22] P. Fortin, Multipole-to-local operator in the fast multipole method: comparison of FFT, rotations and BLAS improvements, Research Report 5752, INRIA, 2005.
[23] M.A. Epton, B. Dembart, Multipole translation theory for the three-dimensional Laplace and Helmholtz equations, SIAM Journal on Scientific Computing 16 (4) (1995) 865–897.
[24] M. Frigo, S.G. Johnson, The design and implementation of FFTW3, Proceedings of the IEEE 93 (2) (2005) 216–231 (Special issue on Program Generation, Optimization, and Platform Adaptation).
[25] C.H. Choi, J. Ivanic, M.S. Gordon, K. Ruedenberg, Rapid and stable determination of rotation matrices between spherical harmonics by direct recursion, Journal of Chemical Physics 111 (1999) 8825–8831.
[26] H. Cheng, V. Rokhlin, N. Yarvin, Nonlinear optimization, quadrature, and interpolation, SIAM Journal on Optimization 9 (4) (1999) 901–923.
[27] N. Yarvin, V. Rokhlin, Generalized gaussian quadratures and singular value decompositions of integral operators, SIAM Journal on Scientific Computing 20 (2) (1998) 699–718.
[28] J. Dongarra, J.D. Croz, S. Hammarling, R.J. Hanson, An extended set of FORTRAN basic linear algebra subprograms, ACM Transactions on Mathematical Software 14 (1) (1988) 1–17.
[29] C.L. Lawson, R.J. Hanson, D.R. Kincaid, F.T. Krogh, Basic linear algebra subprograms for FORTRAN usage, ACM Transactions on Mathematical Software 5 (3) (1979) 308–323.
[30] Y. Hu, S.L. Johnsson, Implementing $O(N)$ *N*-body algorithms efficiently in data-parallel languages, Scientific Programming 5 (4) (1996) 337–364.
[31] C.R. Anderson, An implementation of the fast multipole method without multipoles, SIAM Journal on Scientific and Statistical Computing 13 (4) (1992) 923–947.
[32] X. Sun, N.P. Pitsianis, A matrix version of the fast multipole method, SIAM Review 43 (2) (2001) 289–300.
[33] Fortran 77 reference implementation source code of the BLAS from netlib. <http://www.netlib.org/blas>.
[34] J. Dongarra, I.S. Duff, D.C. Sorensen, H.A. van der Vorst, Numerical Linear Algebra for High-Performance Computers (Software, Environments, Tools), SIAM, Philadelphia, PA, USA, 1998.
[35] B. Kågström, P. Ling, C.V. Loan, GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark, ACM Transactions on Mathematical Software 24 (3) (1998) 268–302.
[36] ATLAS homepage. <http://math-atlas.sourceforge.net/>.
[37] R.C. Whaley, A. Petitet, Minimizing development and maintenance costs in supporting persistently optimized BLAS, Software: Practice and Experience 35 (2) (2005) 101–121.
[38] J. Carrier, L. Greengard, V. Rokhlin, A fast adaptive multipole algorithm for particle simulations, SIAM Journal on Scientific and Statistical Computing 9 (4) (1988) 669–686.
[39] O. Coulaud, P. Fortin, J. Roman, High-performance BLAS formulation of the adaptive fast multipole method, in: T. Simos, G. Maroulis (Eds.), Advances in Computational Methods in Sciences and Engineering 2005, Selected Papers from the International Conference of Computational Methods in Sciences and Engineering (ICCMSE 2005), vol. 4B of Special Volume of the Lecture Series on Computer and Computational Sciences, VSP/Brill, 2005, pp. 1796–1799.