SHARED-MEMORY PARALLELIZATION OF THE SPREADING/GRIDDING STEP IN NON-UNIFORM FAST FOURIER TRANSFORM (NUFFT) AND IN FAST EWALD SUMMATION

ERIK BOSTRÖM

Abstract. In this report we consider shared-memory parallelization of the spreading/gridding step that occurs in the non-uniform fast Fourier transform (NUFFT) and in fast Ewald summation. We analyze the performance of fast Gaussian gridding (FGG) of the sequential NUFFT algorithm and analyze possible parallelization strategies of it using C/C++ and OpenMP. Our serial implementation of FGG demonstrates a speedup exceeding 5 when compared to naive gridding, while the parallel implementation exhibits nearly linear speedup, which further improves with increasing problem size. Our relatively straightforward implementation draws inspiration and parallelization concepts from the Flatiron Institute NUFFT (FINUFFT) code.

1. Introduction

The Fast Fourier Transform (FFT) is a crucial algorithm in digital signal processing and computational mathematics. It efficiently computes the Discrete Fourier Transform (DFT) of a sequence, which represents a signal in the frequency domain. Unlike the direct computation of the DFT, which has a time complexity of $\mathcal{O}(n^2)$, the FFT reduces this complexity to $\mathcal{O}(n \log n)$, where n is the number of data points. This dramatic improvement in efficiency makes the FFT indispensable for a wide range of applications, for instance in solving partial differential equations. Its versatility, speed, and widespread use make the FFT a cornerstone in computational mathematics and engineering.

The FFT, unlike the direct Fourier sum it computes, requires the data points to be uniformly distributed. This is however not always the case in practice. There is a need for so-called non-uniform FFTs (NUFFTs) that can handle non-uniform point distributions. NUFFTs are utilized in various applications, such as magnetic resonance imaging (MRI), X-Ray computed tomography, ultrasound, and radar. Additionally, for our specific interest, they play a crucial role in periodic electrostatic and Stokes problems solved by the fast Ewald summation, whose spectral part is equivalent to a NUFFT.

A NUFFT consists of an interpolation step along with a standard FFT, thus theoretically it has a minimum complexity of $\mathcal{O}(n \log n)$. The interpolation step can be executed in various manners. The conventional approach, as utilized in [2, 1], involves convolving the point sources with a kernel in the physical domain, such as a Gaussian [2] or an "exponential of semicircle" [1], and subsequently recovering the true signal through deconvolution. The convolution part of this process is called spreading or gridding and has a naive time-complexity of $\mathcal{O}(NP^3)$, where P is the support of the spreading for each point in the domain. In this report we consider the possible ways to speed up the spreading/gridding step, both for the serial code, but also, which is the main focus, how to do this efficiently in parallel.

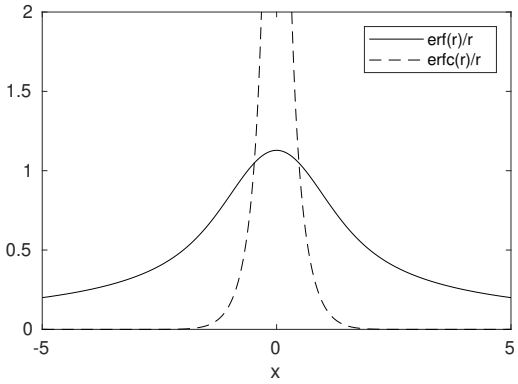

FIGURE 1. *The two components of the Ewald decomposition* (3) *are plotted side by side. The sum of these two terms yields* $1/r$*, where* $r = |x|$.

The report is structured as follows: in Section 2, we provide a brief review of fast Ewald summation and its relationship with the NUFFT; in Section 3, we review the details of the serial NUFFT of our interest – which is the Greengaard and Lee NUFFT [2] – and potential enhancements to the serial algorithm; in Section 4, we outline the algorithms for the parallelization strategies we have implemented; in Section 5, we present a series of numerical tests conducted on our implementations; and in Section 6, we present our conclusions and discuss possible improvements and future continuation of the work.

2. Fast Ewald summation

We consider the computational problem of simulating N charged particles in a periodic primary cell $\Omega = [0, L]^3 \subset \mathbb{R}^3$. We let $\{\mathbf{x}_j\}_{j=1}^N \subset \Omega$ be the locations of the particles and $\{q_j\}_{j=1}^N$ the charges that satisfy charge neutrality $\sum_{j=1}^N q_j = 0$. The system is governed by the singularly forced Poisson equation

$$-\Delta\varphi(\mathbf{x}) = 4\pi \sum_{j=1}^{N} \sum_{\mathbf{p}\in\mathbb{Z}^3} q_j\, \delta(\mathbf{x} - \mathbf{x}_j + L\mathbf{p}), \tag{1}$$

with periodic boundary conditions. This problem has the following fundamental solution:

$$\varphi(\mathbf{x}) = \sum_{j=1}^{N} \sum_{\mathbf{p}\in\mathbb{Z}^3} \frac{q_j}{\|\mathbf{x} - \mathbf{x}_j + L\mathbf{p}\|}. \tag{2}$$

The sums in (2) are however only conditionally convergent. A way of computing truncated sums of this type was proposed by Paul P. Ewald, using a decomposition of the following form:

$$\frac{1}{r} = \frac{\operatorname{erfc}(r)}{r} + \frac{\operatorname{erf}(r)}{r}, \quad r > 0, \tag{3}$$

where $\operatorname{erf}(r)$ is the error function and $\operatorname{erfc}(r)$ its complement. The first term $\operatorname{erfc}(r)/r$ tend more rapidly to zero than $\operatorname{erfc}(r)/r$ as r grows; see Figure 1. And, since the second term is smooth, it converges rapidly in reciprocal space . We utilize this fact when solving the Poisson problem (1). Rather than expressing the solution as (2), we decompose it into two parts as outlined in (3). We handle the first term

in the physical domain and the second term in the frequency domain, where they decay rapidly, respectively.

However, only since the frequency domain part is possible to compute with only a few wavenumbers, it is not in any way an easy task to do efficiently. Evaluating the obtained sums is, in fact, very expensive. Fortunately, fast methods for addressing this problem have been proposed, such as the method by Lindbo and Tornberg [3], which is based on similar ideas as the non-uniform fast Fourier transform. Below, we provide a brief review of the method proposed in [3].

2.1. **Splitting into real- and frequency-domain parts.** To obtain the splitting (3) we start considering the right hand side of the Poisson equation (1). We let

$$-\Delta\varphi(\mathbf{x}) = 4\pi \sum_{j=1}^{N} \sigma_j(\mathbf{x}),$$

where

$$\sigma_j(\mathbf{x}) := \sum_{\mathbf{p}\in\mathbb{Z}^3} q_j\, \delta(\mathbf{x} - \mathbf{x}_j + L\mathbf{p}).$$

Then, we split the σ_j into two parts, one for the real (R) and one for the frequency domain (F) sum:

$$\sigma_j(\mathbf{x}) = \sigma_j^R(\mathbf{x}) + \sigma_j^F(\mathbf{x}),$$

where

$$\sigma_j^R(\mathbf{x}) := \sigma_j(\mathbf{x}) - (\sigma_j * \gamma)(\mathbf{x})$$
$$\sigma_j^F(\mathbf{x}) := (\sigma_j * \gamma)(\mathbf{x}).$$

In the equations above γ is a so called "screening function", which in principle can be any normalized function $\|\gamma(\mathbf{x})\| = 1$ that decays smoothly away from zero. For our purpose however we choose it as a Gaussian

$$\gamma(\mathbf{x}) = \frac{\xi^3}{\pi^{3/2}}\, e^{-\xi^2\|\mathbf{x}\|^2}, \tag{4}$$

which has the Fourier transform

$$\widehat{\gamma}(\mathbf{k}) = e^{-\|\mathbf{k}\|^2/4\xi^2}. \tag{5}$$

The parameter ξ in the Gaussian expression – also known as *the Ewald parameter* – controls the spread (or decay). It is chosen with the accuracy of the numerical computations in mind.

Now, due to the linearity of the Laplacian operator, it is possible to proceed by solving two separate Poisson problems, one for the real part and one for the frequency domain part. For each particle with index $j \in \{1, \ldots, N\}$ we have

$$-\Delta\varphi_j^R(\mathbf{x}) = 4\pi\sigma_j^R(\mathbf{x}), \tag{6}$$

$$-\Delta\varphi_j^F(\mathbf{x}) = 4\pi\sigma_j^F(\mathbf{x}). \tag{7}$$

The solution φ to (1) is then constructed as the sum of the solutions to (6) and (7) as follows:

$$\varphi(\mathbf{x}) = \varphi^R(\mathbf{x}) + \varphi^F(\mathbf{x}) := \sum_{j=1}^{N} \left(\varphi_j^R(\mathbf{x}) + \varphi_j^F(\mathbf{x})\right).$$

Thus, instead of using the fundamental solution (2) directly – that we recall is only conditionally convergent – we instead solve the Poisson problems (6) and (7), which can be done efficiently. The solutions to these problems are presented below in two Lemmas for which the proofs are presented in Appendix A.

Lemma 1 (Real space part of φ)**.** *The sums of the solutions to* (6) *at the particle positions* $\{\mathbf{x}_m\}_{m=1}^N$ *are*

$$\varphi^R(\mathbf{x}_m) = \sum_{j=1}^{N} \sum_{\mathbf{p}\in\mathbb{Z}^3}^{*} q_j \frac{\operatorname{erfc}(\xi\|\mathbf{x}_m - \mathbf{x}_j + L\mathbf{p}\|)}{\|\mathbf{x}_m - \mathbf{x}_j + L\mathbf{p}\|} - q_m \frac{2\xi}{\sqrt{\pi}}, \quad m = 1, 2, \ldots, N, \tag{8}$$

where $*$ *means that the term* $\mathbf{p} = \mathbf{0}$ *is excluded as* $j = m$

Lemma 2 (Frequency domain part of φ)**.** *The sums of the solutions to* (7) *at the particle positions* $\{\mathbf{x}_m\}_{m=1}^N$ *are given by*

$$\varphi^F(\mathbf{x}_m) = \frac{4\pi}{L^3} \sum_{\mathbf{k}\neq 0} \frac{e^{-\|\mathbf{k}\|^2/4\xi^2}}{\|\mathbf{k}\|^2} \sum_{j=1}^{N} q_j\, e^{i\mathbf{k}\cdot(\mathbf{x}_m - \mathbf{x}_j)}, \quad m = 1, 2, \ldots, N. \tag{9}$$

2.2. **Evaluating the sums.** Evaluating (8) or (9) directly requires $\mathcal{O}(N^2)$ operations each. However, significant improvements can be made. For the real sum, for instance, one can, as described in [3], impose a truncation radius r_c such that each particle only interacts with a few of its neighbors. In this context, let $\mathbf{1}_{[0,r_c]}(r)$ denote the indicator function, which is equal to one if r is in the interval $[0, r_c]$ and zero otherwise. By applying the indicator function to (8) we get

$$\varphi^R_{r_c}(\mathbf{x}_m) := \sum_{\mathbf{y}\in\mathcal{N}_m} q(\mathbf{y}) \frac{\operatorname{erfc}(\xi\|\mathbf{x}_m - \mathbf{y}\|)}{\|\mathbf{x}_m - \mathbf{y}\|}, \quad m = 1, 2, \ldots, N,$$

where $\mathcal{N}_m$ is the set of neighhbours to $\mathbf{x}_m$ with charge $q(\mathbf{y})$ for $\mathbf{y} \in \mathcal{N}_m$. The real space sum can be evaluated rapidly by selecting the parameter ξ to be sufficiently small. (The parameter ξ governs the truncation radius.) By appropriately implementing a cutoff, the complexity of the real space part can be reduced to $\mathcal{O}(N)$. The emphasis from this point forward is on the frequency domain part, which poses greater demands.

For the frequency domain part, we introduce another parameter $\eta \in (0, 1)$ that can be used to control the width of the Gaussians. Then we obtain the following result, which is the crucual one.

Lemma 3 (Spectral Ewald summation)**.** *Let*

$$H(\mathbf{x}) := \sum_{j=1}^{N} q_j \left(\frac{2\xi^2}{\pi\eta}\right)^{3/2} \sum_{\mathbf{p}\in\mathbb{Z}^3} e^{-2\xi^2\|\mathbf{x}-\mathbf{x}_j+L\mathbf{p}\|^2/\eta}, \tag{10}$$

$$\widehat{\widetilde{H}}(\mathbf{k}) := \frac{e^{-(1-\eta)\|\mathbf{k}\|^2/4\xi^2}}{\|\mathbf{k}\|^2}\, \widehat{H}(\mathbf{k}). \tag{11}$$

where $\widehat{(\cdot)}$ *denotes the the Fourier transform of* $(\cdot)$*. Then,*

$$\varphi^F(\mathbf{x}_m) = 4\pi \int_\Omega \widetilde{H}(\mathbf{x}) \left(\frac{2\xi^2}{\pi\eta}\right)^{3/2} e^{-2\xi^2\|\mathbf{x}-\mathbf{x}_m+L\mathbf{p}\|^2/\eta}\, d\mathbf{x}, \quad m = 1, 2, \ldots, N. \tag{12}$$

Remark 1. It is essential to recognize that $H(\mathbf{x})$ represents the convolution of a periodic function composed of infinitely many Gaussian pulses with the delta distribution. However, in practice, the infinite sum ($\sum_{\mathbf{p}\in\mathbb{Z}^3}$) can be disregarded because the Gaussians are highly localized and can be truncated to achieve sufficient accuracy, as long as the function H remains periodic. In [3], they extend the domain to $[-w, L + w]$, where w denotes the width of the Gaussians, to accommodate periodicity. Further discussion on this matter will be presented in Section 3.

The steps to evaluate the frequency-domain sum are the following [3]:

(1) Create a uniform grid across the three-dimensional rectangular tripple-periodic region $[0, L]^3$ and evaluate the function $H(\mathbf{x})$ on this grid using equation (10).
(2) Since the grid is evenly spaced, $\widehat{H}$ can be computed efficiently using the three-dimensional fast Fourier transform (FFT).
(3) Apply the scaling factor in (11) to obtain $\widehat{\widetilde{H}}$.
(4) Utilize an inverse FFT to obtain the transformed function $\widetilde{H}(\mathbf{x})$.
(5) Compute the integral (12) using the trapezoidal rule to obtain the final result, $\{\varphi^F(\mathbf{x}_m)\}$.

It is crucial to keep in mind here that the particles $\mathbf{x}_j$ are not necessarily located at the grid-points of the mesh. In fact, the steps above are essentially the steps of a non-uniform FFT (NUFFT).

3. Non-uniform fast Fourier transforms (NUFFTs)

Similar to before, we let $\{\mathbf{x}_j\}_{j=0}^{N-1}$ be the sampling of discrete points in a three dimensional periodic domain, which we define as $\Omega := [0, 2\pi)^3$. We furthermore let f be a periodic and real valued function on Ω and let $f_j := f(\mathbf{x}_j)$ be the source values at the points. Furthermore, let $\mathbf{k} \in S_M = \{(k_1, k_2, k_3) \in \mathbb{Z}^3 \colon -\frac{M}{2} \leq k_1, k_2, k_3 < \frac{M}{2}\}$ be discrete *wavenumbers* in the frequency domain and let $F(\mathbf{k})$ denote the corresponding *Fourier coefficients* of $f(\mathbf{x})$.

The forward and backward (inverse) Fourier transforms of f can now be stated as

$$F(\mathbf{k}) = \frac{1}{N} \sum_{j=0}^{N-1} f_j e^{-i\mathbf{k}\cdot\mathbf{x}_j}, \quad \mathbf{k} \in S_M, \tag{13}$$

$$f(\mathbf{x}_j) = \sum_{\mathbf{k}\in S_M} F(\mathbf{k})\, e^{i\mathbf{k}\cdot\mathbf{x}_j}, \quad j = 0, 1, \ldots, N-1. \tag{14}$$

We follow [2] and call (13) the forward, type I NUFFT, and (14) the backward, type II NUFFT. For the purpose of the spectral Ewald method, we only consider the type I version.

The direct sum (13) is computed in complexity $\mathcal{O}(NM^3)$. When the sampling is uniform and $N = M^3$, it can moreover be computed with the *fast Fourier transform* (FFT) in $\mathcal{O}(N \log N)$. But, if the points are non-uniform, the FFT algorithm does not directly apply. A non-uniform FFT (NUFFT) is based on combining some interpolation scheme with the standard FFT to obtain (13). This result in a complexity that is almost as fast as the original FFT provided the interpolation can be done efficiently.

There are different NUFFT algorithms availible. In this report we consider the simple and efficient NUFFT proposed in Greengard and Lee [2] which is also used by Lindbo and Tornberg [3] to compute the spectral Ewald sum.

3.1. Type I NUFFT (non-uniform to uniform).

Let $\mathbf{x} \in \Omega$. The type I NUFFT (13) describes the Fourier series of the distribution

$$f(\mathbf{x}) = \sum_{j=0}^{n} f_j(\mathbf{x}_j)\delta(\mathbf{x} - \mathbf{x}_j),$$

where $\delta(\mathbf{x} - \mathbf{x}_j)$ is the Dirac measure on $\mathbb{R}^3$ giving unit mass for $\mathbf{x} = \mathbf{x}_j$. Following [2], the idea is now to "replace" the δ distribution by a Gaussian. The Gaussian has a spread so that the contribution of the off-grid-values of f are possible to be taken into account. The convolution with a Gaussian gives a "blurred picture", for

which the true signal later can be recovered by deconvolution. The procedure is described below.

A periodic function of a symmetric Gaussian pulse over Ω with spreading τ can be defined as

$$g_\tau(\mathbf{x}) = \sum_{\mathbf{p}\in\mathbb{Z}^3} e^{-\|\mathbf{x}-2\pi\mathbf{p}\|^2/4\tau}.$$

A new function f_τ are then defined as the convolution of f with g_τ as

$$f_\tau(\mathbf{x}) := (f * g_\tau)(\mathbf{x}),$$

where the Fourier coefficients of f_τ are given by

$$F_\tau(\mathbf{k}) = \frac{1}{(2\pi)^3}\int_\Omega f_\tau(\mathbf{x}) e^{-i\mathbf{k}\cdot\mathbf{x}}\,\mathrm{d}\mathbf{x}.$$

Since f_τ is periodic, the integral can be approximated up to spectral accuracy by the trapetzoidal rule. To minimize aliasing errors this is done on an oversampled grid. Let R be a positive over-sampling ratio that is typically between 1 and 2, then we define $M_r := R\,M$ as the number of oversampling grid points in each direction in the box $\Omega_{M_r} := \{2\pi/M_r(i,j,k)\colon i,j,k = 0,1,\ldots,M_r-1\} \subset \Omega$, and get

$$F_\tau(\mathbf{k}) \approx \frac{1}{M_r^3}\sum_{\boldsymbol{\xi}\in\Omega_{M_r}} f_\tau(\boldsymbol{\xi})\, e^{-i\mathbf{k}\cdot\boldsymbol{\xi}},$$

where

$$f_\tau(\boldsymbol{\xi}) = \sum_{j=0}^{N-1} f(\mathbf{x}_j) g_\tau(\mathbf{x}_j - \boldsymbol{\xi}), \quad \boldsymbol{\xi}\in\Omega_{M_r}, \tag{15}$$

$$= \sum_{j=0}^{N-1} f(\mathbf{x}_j) e^{-\|\mathbf{x}_j-\boldsymbol{\xi}\|_*^2/4\tau}. \quad \boldsymbol{\xi}\in\Omega_{M_r} \tag{16}$$

Here $*$ denotes that periodicity has been applied. As the values of $F_\tau(\mathbf{k})$ have been computed for $\mathbf{k}\in S_M$, one can finally obtain $F(\mathbf{k})$ by the deconvolution

$$F(\mathbf{k}) = F_\tau(\mathbf{k})\, G_\tau(\mathbf{k})^{-1}, \tag{17}$$

where $G_\tau(\mathbf{k})^{-1}$ is the inverse of the Fourier transform of g_τ that is given by

$$G_\tau(\mathbf{k})^{-1} = \left(\frac{\pi}{\tau}\right)^{3/2} e^{\tau\|\mathbf{k}\|^2}.$$

Remark 2. In practice one typically computes $F_\tau(\mathbf{k})$ for $\mathbf{k}\in S_{M_r}$ from f_τ by the FFT and then use $F_\tau(\mathbf{k})$ for $\mathbf{k}\in S_M$ in the deconvolution step, i.e., the modes of $S_{M_r}\setminus S_M$ are thrown away.

Remark 3. Since the Gaussians decay to zero rapidly in (16), one does not need to evaluate them for all $\boldsymbol{\xi}\in\Omega_{M_r}$, but only for the points that are closest to $\mathbf{x}_j$. The number of points that are used in each direction is called the *spreading* of the Gaussian. The number of points used in the spreading control the accuracy of the NUFFT.

Remark 4. For the spectral Ewald method, the H function in (10) is the counterpart to g_τ in (15).

Algorithm 1 Naive gridding

Input: $\{x_j\}_{j=0}^N, \{y_j\}_{j=0}^N, \{z_j\}_{j=0}^N, \{f_j\}_{j=0}^N, M_r, M_{sp}, \tau$
Output: f_τ

for $j \in \{0, 1, \ldots, N-1\}$ **do**
  Find nearest $(\xi_1, \xi_2, \xi_3) = \frac{2\pi}{M_r}(m_1, m_2, m_3)$ with $\xi_i \le x_j$, $i = 1, 2, 3$.
  for $l_1 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
    for $l_2 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
      for $l_3 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
        $(i_1, i_2, i_3) \leftarrow ((m_1, m_2, m_3) + (l_1, l_2, l_3))$ positive mod M_r
        $f_\tau[i_1, i_2, i_3]$ += $f_j e^{-\left((x_j - \xi_1 - \frac{2\pi}{M_r} l_1)^2 + (y_j - \xi_2 - \frac{2\pi}{M_r} l_2)^2 + (z_j - \xi_3 - \frac{2\pi}{M_r} l_3)^2\right)/4\tau}$
      end for
    end for
  end for
end for

3.2. **Fast Gaussian Gridding (FGG).** The most computational intensive parts of the algorithms for type I is the calculations of $f_\tau(\boldsymbol{\xi})$ in (16). The process of computing $f_\tau(\boldsymbol{\xi})$ on Ω_{M_r} is called *gridding*. A naive algorithm of the gridding step of NUFFT type I is presented in Algorithm 1. The naive gridding computes $\{f_\tau(\boldsymbol{\xi})\}$ in $\mathcal{O}(NP^3)$, where $P = 2M_{sp}$ is the full spread across a Gaussian.

By *Fast Gaussian Gridding* (FGG) [2] one speeds up the exponential computations. The concept relies on decomposing the exponential in the g_τ function. This decomposition allows pre-computation of certain parts, reducing both the number of exponential evaluations and the number of multiplications.

For the one-dimensional exponential we have

$$\begin{aligned} e^{-(x-2\pi l/M_r)^2/4\tau} &= e^{-(x^2 - 4\pi x l/M_r + 4\pi^2 l^2/M_r^2)/4\tau} \\ &= e^{-(x^2 - 4\pi x l/M_r + 4\pi^2 l^2/M_r^2)/4\tau} \\ &= e^{-x^2/4\tau + \pi x l/(M_r\tau) - \pi^2 l^2/(M_r^2\tau)} \\ &= e^{-x^2/4\tau} (e^{\pi x/(M_r\tau)})^l e^{-\pi^2 l^2/(M_r^2\tau)}. \end{aligned}$$

Thus, we get the decomposition

$$e^{(x-2\pi l/M_r)^2/4\tau} = E_1(x)(E_2(x))^l E_3(l), \tag{18}$$

where $E_1(x) := e^{-x^2/4\tau}$ and $E_2(x) := e^{-\pi x/(M_r\tau)}$ are independent on l and $E_3(l) := e^{-\pi^2 l^2/(M_r^2\tau)}$ is independent on x.

The Gaussian kernel is build as a tensor product. Hence, the extension of (18) to 2D and 3D is straight forward. Let $\mathbf{l} := (l_1, l_2, l_3)$; in 3D we then have

$$\begin{aligned} e^{-\|\mathbf{x} - 2\pi\mathbf{l}/M_r\|^2/4\tau} &= e^{-(x-2\pi l_1/M_r)^2/4\tau} e^{-(y-2\pi l_2/M_r)^2/4\tau} e^{-(z-2\pi l_3/M_r)^2/4\tau}, \\ &= e^{-(x+y+z)^2/4\tau} \left(e^{x\pi/M_r\tau}\right)^{l_1} \left(e^{y\pi/M_r\tau}\right)^{l_2} \left(e^{z\pi/M_r\tau}\right)^{l_3} \\ &\quad \times e^{-(\pi l_1/M_r)^2/\tau} e^{-(\pi l_2/M_r)^2/\tau} e^{-(\pi l_3/M_r)^2/\tau}, \\ &= E_1(\mathbf{x}) E_2(x)^{l_1} E_2(y)^{l_2} E_2(z)^{l_3} E_3(l_1) E_3(l_2) E_3(l_3), \end{aligned}$$

where

$$\begin{aligned} E_1(x, y, z) &= e^{-(x+y+z)^2/4\tau} \\ E_2(x) &= e^{\pi x/M_r\tau} \\ E_3(l) &= e^{-(\pi l/M_r)^2/\tau}. \end{aligned}$$

Equation (15) can now be approximated as

$$
\begin{aligned}
f_\tau(\boldsymbol{\xi}) &= \sum_{j=0}^{N-1} f(\mathbf{x}_j) g_\tau(\mathbf{x}_j - \boldsymbol{\xi}) \\
&\approx \sum_{j=0}^{N-1} \sum_{l_1=-M_{sp}+1}^{M_{sp}} \sum_{l_2=-M_{sp}+1}^{M_{sp}} \sum_{l_3=-M_{sp}+1}^{M_{sp}} f(\mathbf{x}_j) E_1(\mathbf{x}_j - \boldsymbol{\xi}) \\
&\qquad \times E_2(x_j - \xi_1)^{l_1} E_2(y_j - \xi_2)^{l_2} E_2(z_j - x i_3)^{l_3} E_3(l_1) E_3(l_2) E_3(l_3) \qquad (19) \\
&= \sum_{j=0}^{N-1} f(\mathbf{x}_j) E_1(\mathbf{x}_j - \boldsymbol{\xi}) \Bigg(\sum_{l_1=-M_{sp}+1}^{M_{sp}} E_2(x_j - \xi_1)^{l_1} E_3(l_1) \\
&\qquad \Bigg(\sum_{l_2=-M_{sp}+1}^{M_{sp}} E_2(y_j - \xi_2)^{l_2} E_3(l_2) \Bigg(\sum_{l_3=-M_{sp}+1}^{M_{sp}} E_2(z_j - \xi_3)^{l_3} E_3(l_3) \Bigg) \Bigg) \Bigg). \qquad (20)
\end{aligned}
$$

By pre-computing the exponentials $E_2(x_j - \xi_i)^{l_i} E_3(l_i)$, $i = 1, 2, 3$, for $l_i = -M_{sp} + 1, \ldots, M_{sp}$, the rewriting (19) into (20) decreases the number of arithmetic operations of the spreading significantly. The "naive" sum (19) requires $7P^3$ multiplications and P^3 additions, but (20) requires only $4P^3$ multiplications and P^3 additions.

Pseudo-code for two versions of the fast Gaussian gridding algorithm is presented in Algorithms 2–3. The first version is a naive implementation of the FGG, considering only the storage aspect. In the second version, we have further optimized the algorithm by reducing the number of multiplications inside the nested loops of the spreading phase.

Algorithm 2 Naive Fast Gaussian Gridding

Input: $\{x_j\}_{j=0}^N, \{y_j\}_{j=0}^N, \{z_j\}_{j=0}^N, \{f_j\}_{j=0}^N, M_r, M_{sp}, \tau$
Output: f_τ

Precompute $E_3(i)$ for $i \in \{-M_{sp}+1, \ldots, M_{sp}\}$.
for $j \in \{0, 1, \ldots, N-1\}$ **do**
  Find nearest $(\xi_1, \xi_2, \xi_3) = \frac{2\pi}{M_r}(m_1, m_2, m_3)$ with $\xi_i \leq x_j$, $i = 1, 2, 3$.
  $E_1 \leftarrow E_1(x_j - \xi_1, y_j - \xi_2, z_j - \xi_3)$
  $E_{2x}(l) \leftarrow E_2(x_j - \xi_1)^l$ for $l \in \{-M_{sp}+1, \ldots, M_{sp}\}$
  $E_{2y}(l) \leftarrow E_2(y_j - \xi_2)^l$ for $l \in \{-M_{sp}+1, \ldots, M_{sp}\}$
  $E_{2z}(l) \leftarrow E_2(x_j - \xi_3)^l$ for $l \in \{-M_{sp}+1, \ldots, M_{sp}\}$
  for $l_1 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
    for $l_2 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
      for $l_3 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
        $(i_1, i_2, i_3) \leftarrow ((m_1, m_2, m_3) + (l_1, l_2, l_3))$ positive mod M_r
        $f_\tau[i_1, i_2, i_3]$ += $f_j\, E_1\, E_{2x}(l_1)\, E_{2y}(l_2)\, E_{2z}(l_3)\, E_3(l_1)\, E_3(l_2)\, E_3(l_3)$
      end for
    end for
  end for
end for

Algorithm 3 Improved Fast Gaussian Gridding

Input: $\{x_j\}_{j=0}^N, \{y_j\}_{j=0}^N, \{z_j\}_{j=0}^N, \{f_j\}_{j=0}^N, M_r, M_{sp}, \tau$
Output: f_τ

Precompute $E_3(i)$ for $i \in \{-M_{sp}+1, \ldots, M_{sp}\}$.
for $j \in \{0, 1, \ldots, N-1\}$ **do**
  Find nearest $(\xi_1, \xi_2, \xi_3) = \frac{2\pi}{M_r}(m_1, m_2, m_3)$ with $\xi_i \le x_j$, $i = 1, 2, 3$.
  $E_1 \leftarrow E_1(x_j - \xi_1, y_j - \xi_2, z_j - \xi_3)$
  $E_x(l) \leftarrow E_2(x_j - \xi_1)^l E_3(l_1)$ for $l \in \{-M_{sp}+1, \ldots, M_{sp}\}$
  $E_y(l) \leftarrow E_2(y_j - \xi_2)^l E_3(l_2)$ for $l \in \{-M_{sp}+1, \ldots, M_{sp}\}$
  $E_z(l) \leftarrow E_2(x_j - \xi_3)^l E_3(l_3)$ for $l \in \{-M_{sp}+1, \ldots, M_{sp}\}$
  $V_0 \leftarrow f_j\, E_1$
  for $l_1 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
    $V_x \leftarrow V_0\, E_x(l_1)$
    $i_1 \leftarrow (m_1 + l_1)$ positive mod M_r
    for $l_2 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
      $V_y \leftarrow V_x\, E_y(l_2)$
      $i_2 \leftarrow (m_2 + l_2)$ positive mod M_r
      for $l_3 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
        $i_3 \leftarrow (m_3 + l_3)$ positive mod M_r
        $f_\tau[i_1, i_2, i_3]$ += $V_y\, E_z(l_3)$
      end for
    end for
  end for
end for

4. Parallellization strategies

We now move on and consider parallelization strategies of the FGG algorithm (Algorithm 3). We use the `OpenMP` interface for the C/C++ programming language.

Recall that the computational domain is the three-dimensional box $\Omega = [0, 2\pi)^3$. Each particle at position $\mathbf{x}_j$ has a spread of P^3 points around it, where $P = 2M_{sp}$. The array to be evaluated is $\{f_\tau(\boldsymbol{\xi})\}$. The array is updated at all points inside the spreading box. If two particles are located sufficiently close to each other so that their spreading regions overlap, the charges of both particles will contribute to f_τ at the points of intersection. An illustrative drawing of this is presented in Figure 2.

In order to parallelize the serial FGG algorithm we can immediately notice the following issues:

- we have a reduction operation (+=) on $f_\tau(\boldsymbol{\xi})$;
- $\{f_\tau(\boldsymbol{\xi})\}$ is a shared array to write to;
- the non-uniform points $\{\mathbf{x}_j\}$ are scattered over the domain, typically in random order;
- if we parallelize the outer loop so that each thread takes care of a certain number of points, there will be race conditions for all points in overlapping spreading regions (see Figure 2).
- a critical region for the update of $f_\tau(\boldsymbol{\xi})$ will obviously lead to synchronization overhead.

These issues make a simple OpenMP parallelization strategy inefficient. Firstly, we aggregate the elements of an array, not a variable. Secondly, having many points in the domain will lead to severe race conditions due to multiple overlapping spreading regions.

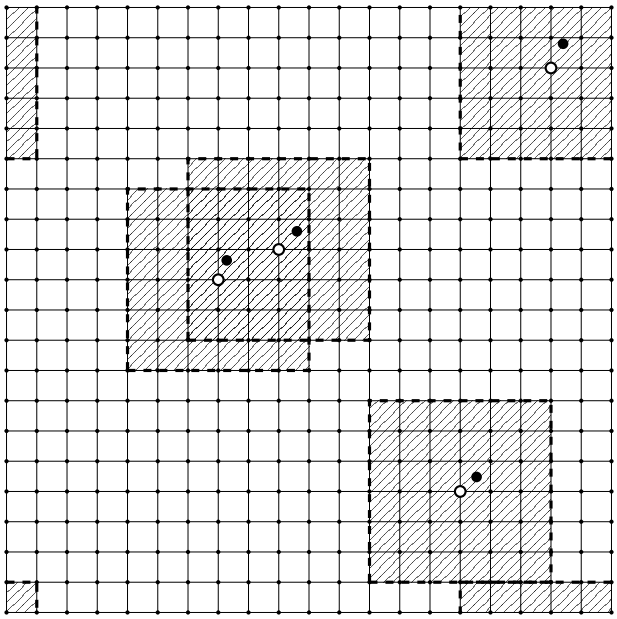

FIGURE 2. *In this figure we show a 2D illustration of the oversampled grid – it may as well serve as an illustration for the 3D problem when viewed as a cross-section. As an example the grid includes four off-grid points, represented by filled black circles. The white points in the illustration denote the closest neighbors, whose indices are closer than one unit lower than the off-grid coordinates of their corresponding off-grid points. The chosen spreading parameter for this example is* $M_{sp} = 3$*, resulting in the regions delineated by dashed lines. It is important to note that there is overlap in the regions of two points. Values obtained within this overlapping region contribute to the* $\{f_\tau(\boldsymbol{\xi})\}$ *array for both points. Additionally, the point in the upper right corner exhibits a spreading region that extends beyond the grid boundaries. In this case, given the triple periodic nature of the domain, the spreading region continues periodically on the opposite side of the domain in all periodic directions.*

In a first, naive attempt to parallelize Algorithm 3 using OpenMP, we let each thread spread the result to a local array $\{f_\tau^{\text{loc}}(\boldsymbol{\xi})\}$ of full size N. Once the spreading is completed, each element of the local array is added to the global array inside a critical region; see Algorithm 4.

The naive version of the parallel algorithm has however a severe performance problem. Increasing the number of threads do certainly speed-up the main loop, but the complexity of the critical region is still $\mathcal{O}(N)$. We can expect the synchronization time of the critical region to increase close to linear with the number of threads. For a big grid, the synchronization overhead would be of substantial size.

To enhance the naive algorithm, we initiate the process by sorting the non-uniform points into equally large subsets, each corresponding to a sub-region of the domain. For this purpose, we employ an unfinished bin-sort algorithm, akin to the one used in [1]. The unfinished bin-sort organizes the points in small chunks of size $16\times4\times4$ that fits into the cache to maximize the amount of cache hits. These chunks are moreover combined into one subset for each thread. As the non-uniform points are sorted, the smallest and largest indices in each dimension are stored to establish the limits for the local subregions of the oversampled grid Ω_{M_r}. Additionally, the

Algorithm 4 Naive parallel fast Gaussian gridding

Input: $\{x_j\}_{j=0}^N, \{y_j\}_{j=0}^N, \{z_j\}_{j=0}^N, \{f_j\}_{j=0}^N, M_r, M_{sp}, \tau$
Output: f_τ

Precompute $E_3(l)$ for $l \in \{-M_{sp}+1, \ldots, M_{sp}\}$.
#**pragma omp parallel**{
#**pragma omp for**
for $j \in \{0, 1, \ldots, N-1\}$ **do**
    Find nearest $(\xi_1, \xi_2, \xi_3) = \frac{2\pi}{M_r}(m_1, m_2, m_3)$ such that $\xi_i \leq x_j$, $i = 1, 2, 3$.
    $E_1 \leftarrow E_1(x_j - \xi_1, y_j - \xi_2, z_j - \xi_3)$
    $E_x(l) \leftarrow E_2(x_j - \xi_1)^l E_3(l)$ for $l \in \{-M_{sp}+1, \ldots, M_{sp}\}$
    $E_y(l) \leftarrow E_2(y_j - \xi_2)^l E_3(l)$ for $l \in \{-M_{sp}+1, \ldots, M_{sp}\}$
    $E_z(l) \leftarrow E_2(x_j - \xi_3)^l E_3(l)$ for $l \in \{-M_{sp}+1, \ldots, M_{sp}\}$
    $V_0 \leftarrow f_j\, E_1$
    for $l_1 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
        $V_x \leftarrow V_0\, E_x(l_1)$
        $i_1 \leftarrow (m_1 + l_1)$ positive mod M_r
        for $l_2 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
            $V_y \leftarrow V_x\, E_y(l_2)$
            $i_2 \leftarrow (m_2 + l_2)$ positive mod M_r
            for $l_3 \in \{-M_{sp}+1, \ldots, M_{sp}\}$ **do**
                $i_3 \leftarrow (m_3 + l_3)$ positive mod M_r
                $f_\tau^{\mathrm{loc}}[i_1, i_2, i_3]$ += $V_y\, E_z(l_3)$
            end for
        end for
    end for
end for
#**pragma omp critical**
for $m \in \{0, 1, \ldots, M_r\, M_r\, M_r - 1\}$ **do**
    $f_\tau[m]$ += $f_\tau^{\mathrm{loc}}[m]$
end for}

subregions are expanded by M_{sp} points in each direction to accommodate for the spreading.

The primary loop of the algorithm is now improved by increasing the number of cache hits. Furthermore, there is no need to account for periodicity in the main loop since the subregions have their own indexing, hence also the modulo operations that were used in that regard can be removed, which decrease the number of clock-cycles further.

Once the parallel spreading is completed, the local contributions are added to the main $\{f_\tau(\boldsymbol{\xi})\}$ array in a critical region where the number of iterations of the for loop now scales with the number of threads. The algorithm is presented in Algorithm 5.

Algorithm 5 Improved parallel fast Gaussian gridding

Input: $\{x_j\}_{j=0}^N, \{y_j\}_{j=0}^N, \{z_j\}_{j=0}^N, \{f_j\}_{j=0}^N, M_r, M_{sp}, \tau$
Output: f_τ

$\{\text{indices}\} \leftarrow \text{bin\_sort}(\{x_j\}_{j=0}^N, \{y_j\}_{j=0}^N, \{z_j\}_{j=0}^N)$
#**pragma omp parallel**{
$E_3(i,j,k) \leftarrow E_3(i)E_3(j)E_3(k)$ for $i,j,k \in \{-M_{sp}+1,\ldots,M_{sp}\}$.
$n_p \leftarrow \text{omp\_get\_num\_threads}()$
#**pragma omp for**{
for $i_p \in \{0,1,\ldots,n_p-1\}$ **do**
  $N^{\text{loc}} \leftarrow \text{get\_nr\_of\_points}(i_p, n_p)$
  $(\{x_j^{\text{loc}}\}, \{y_j^{\text{loc}}\}, \{z_j^{\text{loc}}\}, \{f_j^{\text{loc}}\}) \leftarrow \text{get\_local}(\{\text{indices}\}, \{x_j\}, \{y_j\}, \{z_j\}, \{f_j\})$
  $(o_x, o_y, o_z, M_x, M_y, M_z) \leftarrow \text{get\_subgrid}(i_p, \{x_j^{\text{loc}}\}, \{y_j^{\text{loc}}\}, \{z_j^{\text{loc}}\})$
  for $j \in \{0,1,\ldots,N^{\text{loc}}-1\}$ **do**
    Find nearest $(\xi_1,\xi_2,\xi_3) = \frac{2\pi}{M_r}(m_1,m_2,m_3)$ with $\xi_i \le x_j$, $i=1,2,3$.
    $E_1 \leftarrow E_1(x_j-\xi_1, y_j-\xi_2, z_j-\xi_3)$
    $E_x(l) \leftarrow E_2(x_j-\xi_1)^l E_3(l)$ for $l \in \{-M_{sp}+1,\ldots,M_{sp}\}$
    $E_y(l) \leftarrow E_2(y_j-\xi_2)^l E_3(l)$ for $l \in \{-M_{sp}+1,\ldots,M_{sp}\}$
    $E_z(l) \leftarrow E_2(x_j-\xi_3)^l E_3(l)$ for $l \in \{-M_{sp}+1,\ldots,M_{sp}\}$
    $V_0 \leftarrow f_j\, E_1$
    for $l_1 \in \{-M_{sp}+1,\ldots,M_{sp}\}$ **do**
      $Vx \leftarrow V_0\, E_x(l_1)$
      $i_1 \leftarrow m_1 + l_1 - o_x$
      for $l_2 \in \{-M_{sp}+1,\ldots,M_{sp}\}$ **do**
        $V_y \leftarrow V_x\, E_y(l_2)$
        $i_2 \leftarrow m_2 + l_2 - o_y$
        for $l_3 \in \{-M_{sp}+1,\ldots,M_{sp}\}$ **do**
          $i_3 \leftarrow m_3 + l_3 - o_z$
          $f_\tau^{\text{loc}}[i_1,i_2,i_3] \mathrel{+}= V_y\, E_z(l_3)$
        end for
      end for
    end for
  end for
  #**pragma omp critical**
  $\{f_\tau\} \leftarrow \text{add\_subproblem\_result}(\{f_\tau^{\text{loc}}\}, o_x, o_y, o_z, M_x, M_y, M_z)$
end for

5. Numerical results

Numerical tests were run on pseudo-random data. The N points $\{\mathbf{x}_j\} \in \Omega$ as well as random point source values $\{f_j\}$ were created using the `rand()` routine in the C language.

The machine that we used to run the tests was a desktop computer with a 12 core AMD Ryzen 9 CPU of 3.8 GHz, running Ubuntu 22.04.3 LTS. Using multithreading it is possible to run the CPU on 24 threads; however, the performance of the multithreading using 13 to 24 threads showed a significant performance decrease and we therefore restricted the analysis to 12 threads in maximum. The timings presented are the shortest times of a set of runs, making the comparision as accurate as possible.

5.1. **Serial FGG performance.** In Figure 3 and Table 1, the computation speed of the naive gridding is compared against the naive FGG (where the entire exponentials are summed together inside the inner loop) and the original FGG (Algorithm

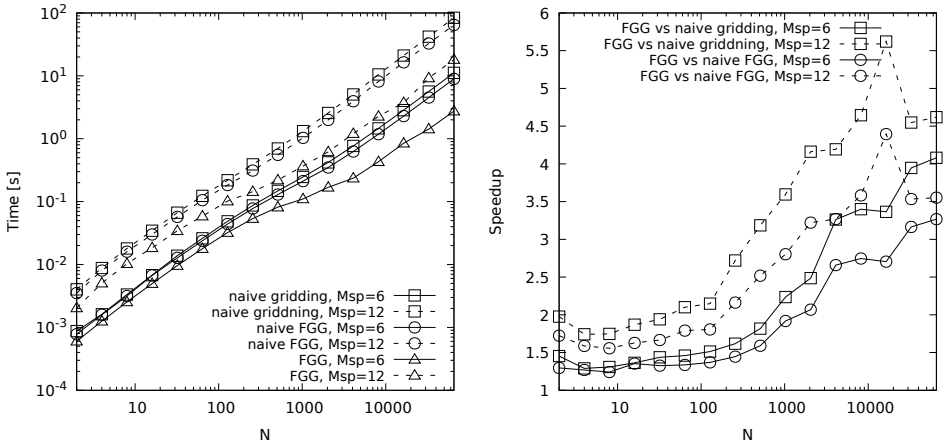

FIGURE 3. *Comparison – naive gridding vs. naive fast Gaussian gridding and the improved Gaussian gridding for two different values of M_{sp}. (left) timing; (right) speedup of the faster FGG algorithm versus the two naive ones. In all simulations, the number of grid points in the uniform grid was chosen as $M = 1000$.*

TABLE 1. *Comparison – naive gridding vs. naive fast Gaussian gridding and the improved Gaussian gridding. Timings (in seconds) and speedup ratios for different problem sizes N, where $M = 1000$ and $M_{sp} = 12$ were held fixed. "Naive/FGG" means the speedup of the improved FGG vs. the naive algorithm, and "NFGG/FGG" means the speedup of the improved FGG vs. the naive FGG algorithm.*

N	Naive	Naive FGG	FGG	Naive/FGG	NFGG/FGG
2	0.004027	0.003511	0.002037	1.723549	1.976943
4	0.008796	0.008016	0.005048	1.587919	1.742455
8	0.017999	0.016030	0.010300	1.556330	1.747506
16	0.034510	0.030048	0.018458	1.627895	1.869644
32	0.066688	0.057262	0.034374	1.665857	1.940073
64	0.123526	0.105344	0.058809	1.791290	2.100461
128	0.218641	0.183746	0.101754	1.805788	2.148723
256	0.391878	0.311350	0.144058	2.161283	2.720280
512	0.700756	0.553851	0.219978	2.517756	3.185571
1024	1.323339	1.031947	0.368059	2.803753	3.595452
2048	2.573884	1.992382	0.618440	3.221626	4.161898
4096	5.061948	3.946154	1.207088	3.269152	4.193520
8192	10.540183	8.131523	2.269264	3.583331	4.644758
16384	21.033079	16.441318	3.741341	4.394499	5.621802
32768	41.999513	32.632711	9.233853	3.534030	4.548428
65536	83.875421	64.510695	18.159415	3.552465	4.618839

3). Additionally, Figure 3 displays the speedup of the improved FGG compared to the naive algorithm and to the naive FGG. Choosing the number of points in the spreading as $M_{sp} = 12$ (which is the default), the FGG algorithm becomes approximately five times faster than the naive one when the number of non-uniform grid points N is large (in the order of 10^3 or larger). For even larger N the improvement in speedup is only marginal. The enhancement lies in the spreading; thus, a larger M_{sp} results in a more significant improvement of the FGG. As an example, for the case of $M_{sp} = 6$, the speedup is only approximately 4. Changing the number of

TABLE 2. *Serial run-times in seconds of the parallel against the serial FGG algorithms.*

N	FGG	Naive parallel FGG	Improved parallel FGG
10^4	2.17	2.32	1.78
10^5	22.20	21.02	14.70
10^6	217.78	213.44	144.20

grid points in the uniform grid M does however not influence the complexity of the algorithms (this has also been tested, but not presented since the differences are neglectible).

Interestingly, as shown in Table 1, the timings of the naive FGG are closer to those of the fully naive algorithm than to those of the improved FGG. This suggests that the significant improvement is primarily a result of reducing the number of multiplications by precomputing the exponentials inside the nested for loop and restricting the modulo operations, rather than in the evaluations of the exponentials themselves.

5.2. **Performance of the parallel implementations.** Parallel implementations typically includes several extra constructs that make them slower than the serial code when running on one thread. If the parallel code is too poor on one thread it doesn't really matter if the speedup is perfect; the performance would be poor anyway. Before we study the parallel scaling, it is therefore of paramount importance to check the serial performance of the parallel algorithms.

The serial naive parallel implementation, presented in Algorithm 4 is identical to the serial FGG algorithm except for the critical region and the spreading to a local array. However, the improved parallel version have several parts – such as sorting and calculating offsets – that possibly could take extra time over the serial code. In Table 2 the timings of the parallel codes running on one thread vs the serial FGG code are presented. Interestingly, the parallel implementation is much faster than the serial code also on one thread. This is true for both for smaller and larger problems. Even the naive parallel version is a tiny bit faster than the serial code for rather large N. One difference between the serial code and the naive parallel code is that the parallel code spreads to a local array and then updates the global array in an additional step. Accessing, reading and writing to memory differ. However, the differences here in time are only marginal. What is more interesing is that the improved parallel implementation indeed runs much faster on one thread than the serial code. However, this is not that surprising. The sorting makes less updates of cache lines. Also, the parallel version does not use any modulo operations inside the spreading loop. Furthermore, it also writes to a local array first, similar to the naive parallel version.

Timing and speedup graphs of the naive and the improved parallel algorithms are presented in Figure 4. The improved algorithm runs both faster and has better speedup performance than the naive one. It is clear from the graphs that the naive algorithm suffer from an increased parallel overhead as the number of threads are increased. For sufficiently small N the times also start to grow as the number of threads are large enough. For the improved algorithm we get close to linear speedup as N is large ($N = 10^5$) and a clear improvement with larger number of threads also for smaller N ($N = 10^4$). Tests for even larger problems, for which the timings are not presented in this report, also indicates that the pattern remains – the speedup improves with N.

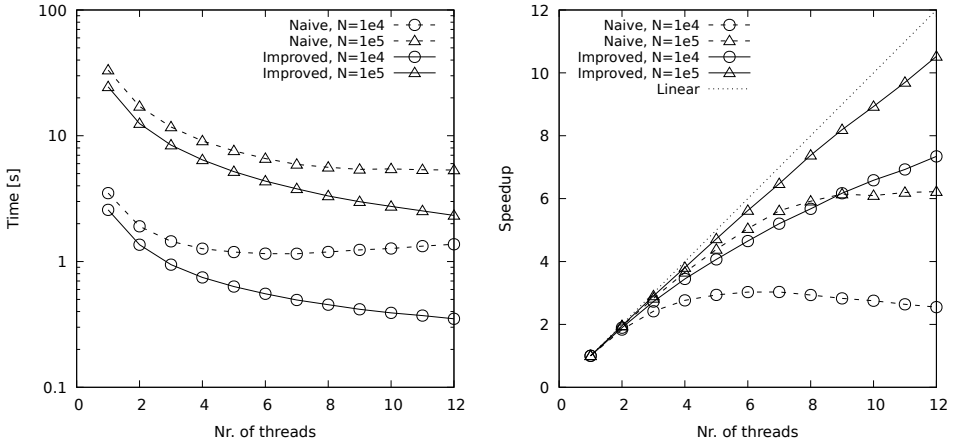

FIGURE 4. *(left) timings of parallel runs; (right) speedup of the parallel runs. For all tests we used* $M_{sp} = 12$ *and* $M = 1000$.

6. Discussion and conclusions

In this report we have analyzed cumputation strategies to speedup the gridding step in 3D NUFFT [2], that for our interest can be used in fast Ewald summation in solving e.g., electrostatic problems [3]. We first analyzed the improvement of the serial algorithm using fast Gaussian gridding (FGG), then we moved on and parallelized the FGG code using similar strategies as in FINUFFT [1].

Using FGG we could speedup the serial version of the NUFFT algorithm with a factor of approximately five. In the original paper of Greengaard and Lee [2] they write that one could expect an improvement of around 5 to 10. However, as the paper is from 2004 the computer architecture as well as compilers has improved and the number of clock-cycles computing an exponential has decreased. We therefore do not expect as great speedup as they got. However, there are still some improvements that can be done. The "positive mod" functions we use employ two modulo operations each which are slow. In the improved parallel version we could also see an improvement over the serial code, that partly could depend on the fact that the mod operations were removed. An interesting observation however is that the pre-storage of the exponentials is not the demanding part of the improvement. Our naive FGG version that precomputes the exponentials but do not pre-compute values inside the nested loop is almost as slow as the naive original code. This fact could also be explained theoretically as the arithmetic operations of the spreading becomes decreased by a factor $7/4 = 1.75$. Other minor improvements that could be considered is the use of the `pow` function and the dynamic vs static allocation of the arrays, etc.

The parallelizations of the FGG algorithm were performed with OpenMP. The big issue we faced with the parallelization of the serial FGG algorithm was the need to write to a shared array on unsorted indices. In a first naive implementation we allocated a full array for each thread. In that way the main loop could run in parallel without any significant parallel overhead. However, after the array had been updated with the local contribution, all threads had to update the original array, that we in the naive version did inside a critical region. Since the local arrays were of full size N the loop inside the critical region required N iterations. Thus, it was expected that the parallel overhead for this version increased both with larger

data and with an increase in the number of threads used. This algorithm is not useful, but it was interesting to consider for the analysis.

To speed up the naive version we took insiration from the ideas used in the Flatron institute NUFFT (FINUFFT) [1] and sorted the N points into equally large subsets, giving subregions of the full domain with offsets given from the smallest and largest coordinates of the local subsets. For the sorting we employed an unfinished bin-sort proposed in [1], that was designed to eliminate unneccessary read and write to RAM. The spreading into the points in the sub-regions did not need to take periodicity of the domain into account; by removing the offets of the indices, each local domain could start at index 0. The improved parallel code also included a critical region; however the amount of work in each subregion was now scaled down with number of threads, making the parallel overhead very small for large N.

Possible improvements to make the code even faster are probably a few – at least on the minor improvement level. One possible improvement that is also implemented in FINUFFT is to "thread-safe" what is now the critical region, using the `#pragma omp atomic` clause. For the atomic clause to work we need to replace the C++ complex type for the arrays that we use in the current version of the code.

Another parallelization strategy that was discussed but not implemented is to split the oversampled Ω_{M_r} grid into disjoint partitions, one for each tread and then sort the source points into each partition with possible duplications for the points that lie within a spreading distance away from the boundary points. The advantage of this strategy is the thread-safe updating of the convolved array. However, by fixing the partition size of the output array, one does not take the number of source points into account. It is possible that the number of points in each partition is severely unbalanced, resulting in load-inbalance.

Appendix A. Proofs of the Lemmas 1–3

Proof of Lemma 1. Let $\Phi(\mathbf{x}) := 1/(4\pi\|\mathbf{x}\|)$ be the fundamental solution to Laplace's equation. Then the solution to (6) can be written as

$$\begin{aligned}
\varphi_j^R(\mathbf{x}) &= \int_{\mathbb{R}^3} \Phi(\mathbf{y}-\mathbf{x}_j)\, 4\pi\, \sigma_j^R(\mathbf{y})\, \mathrm{d}\mathbf{y} \\
&= \int_{\mathbb{R}^3} \frac{1}{4\pi\|\mathbf{y}-\mathbf{x}_j\|} 4\pi \left(\sigma_j(\mathbf{y}) - (\sigma_j * \gamma)(\mathbf{y})\right)\, \mathrm{d}\mathbf{y} \\
&= \int_{\mathbb{R}^3} \frac{1}{\|\mathbf{y}-\mathbf{x}_j\|} \sum_{\mathbf{p}\in\mathbb{Z}^3} \left(q_j\delta(\mathbf{y}-\mathbf{x}_j+L\mathbf{p}) - q_j\gamma(\mathbf{y}-\mathbf{x}_j+L\mathbf{p})\right)\, \mathrm{d}\mathbf{y} \\
&= \sum_{\mathbf{p}\in\mathbb{Z}^3} \int_{\mathbb{R}^3} \frac{q_j}{\|\mathbf{y}-\mathbf{x}_j\|} \left(\delta(\mathbf{y}-\mathbf{x}_j+L\mathbf{p}) - \gamma(\mathbf{y}-\mathbf{x}_j+L\mathbf{p})\right)\, \mathrm{d}\mathbf{y} \\
&= \sum_{\mathbf{p}\in\mathbb{Z}^3} q_j \frac{\operatorname{erfc}(\xi\|\mathbf{x}-\mathbf{x}_j+L\mathbf{p}\|)}{\|\mathbf{x}-\mathbf{x}_j+L\mathbf{p}\|}, \quad \mathbf{x}\neq\mathbf{x}_j.
\end{aligned}$$

(The details of the last step is left for the reader.) For the case $\mathbf{x}\to\mathbf{x}_j$ we have $\operatorname{erfc}(r)/r - 1/r = -\operatorname{erf}(r)/r$, thus it follows that $\varphi_j^R(\mathbf{x}) \to -q_j\frac{2\xi}{\sqrt{\pi}}$ as $\mathbf{x}\to\mathbf{x}_j$. □

Proof of Lemma 2. Expanding φ_j^F in Fourier series, we have

$$\varphi_j^F(\mathbf{x}) = \sum_{\mathbf{k}} \widehat{\varphi}_j^F(\mathbf{k})\, e^{i\mathbf{k}\cdot(\mathbf{x}-\mathbf{x}_j)}$$

The left hand side of (7) can therefore be rewritten as

$$-\Delta\varphi_j^F = \sum_{\mathbf{k}} \|\mathbf{k}\|^2 \widehat{\varphi}_j^F(\mathbf{k})\, e^{i\mathbf{k}\cdot(\mathbf{x}-\mathbf{x}_j)}$$

Using the Poisson summation formula[1] and (4) we can now rewrite the right hand side of (7) as

$$\begin{aligned}4\pi\sigma_j^F &= 4\pi \sum_{\mathbf{p}\in\mathbb{Z}^3} q_j\gamma(\mathbf{x}-\mathbf{x}_j+L\mathbf{p}) \\ &= \frac{4\pi q_j}{L^3}\sum_{\mathbf{k}} \widehat{\gamma}(\mathbf{k})\, e^{i\mathbf{k}\cdot(\mathbf{x}-\mathbf{x}_j)} \\ &= \frac{4\pi q_j}{L^3}\sum_{\mathbf{k}} e^{-\|\mathbf{k}\|^2/4\xi^2}\, e^{i\mathbf{k}\cdot(\mathbf{x}-\mathbf{x}_j)}.\end{aligned}$$

Hence, (7) is equivalent to

$$\sum_{\mathbf{k}} \widehat{\varphi}_j^F(\mathbf{k})\|\mathbf{k}\|^2\, e^{i\mathbf{k}\cdot(\mathbf{x}-\mathbf{x}_j)} = \sum_{\mathbf{k}} \frac{4\pi q_j}{L^3} e^{-\|\mathbf{k}\|^2/4\xi^2}\, e^{i\mathbf{k}\cdot(\mathbf{x}-\mathbf{x}_j)}.$$

This equation implies the following expression for the Fourier-coefficients:

$$\widehat{\varphi}_j^F(\mathbf{k}) = \frac{4\pi q_j}{L^3\|\mathbf{k}\|^2} e^{-\|\mathbf{k}\|^2/4\xi^2}, \quad \mathbf{k}\neq 0.$$

Plugging this back into initial expansion we get

$$\varphi_j^F(\mathbf{x}) = \frac{4\pi}{L^3}\sum_{\mathbf{k}\neq 0} \frac{q_j}{L^3} e^{-\|\mathbf{k}\|^2/4\xi^2}\, e^{i\mathbf{k}\cdot(\mathbf{x}-\mathbf{x}_j)}.$$

Then, the final result follows by summing these solutions. □

Proof of Lemma 3. The function H is the convolution with the delta distribution; hence

$$H(\mathbf{x}) = \sum_{j=1}^N q_j \int_\Omega \delta(\mathbf{y}-\mathbf{x}_j)\left(\frac{2\xi^2}{\pi\eta}\right)^{3/2} e^{-2\xi^2\|\mathbf{y}-\mathbf{x}+L\mathbf{p}\|^2/\eta}\,\mathrm{d}\mathbf{y}.$$

By the convolution theorem a convolution in the physical domain becomes a product in the frequency domain; the Fourier transform of H therefore is

$$\widehat{H}(\mathbf{k}) = \sum_{j=1}^N q_j e^{-\eta\|\mathbf{k}\|^2/8\xi^2}\, e^{-i\mathbf{k}\cdot\mathbf{x}_j}.$$

[1]Poisson summation formula: if we let $f(\mathbf{x})$ have Fourier transform $\hat{f}$, and let $L\neq 0$, then,

$$\sum_{\mathbf{p}\in\mathbb{Z}^3} f(\mathbf{x}+L\mathbf{p}) = \frac{1}{L_3}\sum_{\mathbf{k}} \hat{f}(\mathbf{k})\, e^{i\mathbf{k}\cdot\mathbf{x}}.$$

The frequency domain solution now becomes

$$\begin{aligned}
\varphi^F(\mathbf{x}_m) &= \sum_{j=1}^{N} \varphi_j^F(\mathbf{x}_l) \\
&= \sum_{j=1}^{N} \frac{4\pi q_j}{L^3} \sum_{\mathbf{k}\neq 0} \frac{e^{-\|\mathbf{k}\|^2/4\xi^2}}{\|\mathbf{k}\|^2} e^{i\mathbf{k}\cdot(\mathbf{x}_m-\mathbf{x}_j)} \\
&= \frac{4\pi}{L^3} \sum_{\mathbf{k}\neq 0} \frac{e^{-\|\mathbf{k}\|^2/4\xi^2}}{\|\mathbf{k}\|^2} e^{-i\mathbf{k}\cdot\mathbf{x}_m} \sum_{j=1}^{N} q_j e^{i\mathbf{k}\cdot\mathbf{x}_j} \\
&= \frac{4\pi}{L^3} \sum_{\mathbf{k}\neq 0} \frac{e^{-(1-\eta)\|\mathbf{k}\|^2/4\xi^2}}{\|\mathbf{k}\|^2} e^{-i\mathbf{k}\cdot\mathbf{x}_m} \sum_{j=1}^{N} q_j e^{-\eta\|\mathbf{k}\|^2/4\xi^2} e^{i\mathbf{k}\cdot\mathbf{x}_j} \\
&= \frac{4\pi}{L^3} \sum_{\mathbf{k}\neq 0} e^{-\eta\|\mathbf{k}\|^2/8\xi^2} e^{-i\mathbf{k}\cdot\mathbf{x}_m} \frac{e^{-(1-\eta)\|\mathbf{k}\|^2/4\xi^2}}{\|\mathbf{k}\|^2} \sum_{j=1}^{N} q_j e^{-\eta\|\mathbf{k}\|^2/8\xi^2} e^{i\mathbf{k}\cdot\mathbf{x}_j}
\end{aligned}$$

Hence, we have

$$\varphi^F(\mathbf{x}_l) = \frac{4\pi}{L^3} \sum_{\mathbf{k}\neq 0} e^{-\eta\|\mathbf{k}\|^2/8\xi^2} \widehat{\widetilde{H}}(-\mathbf{k})$$

Now, applying Parseval's theorem, equation (12) is obtained. □

References

[1] Alexander H Barnett, Jeremy Magland, and Ludvig af Klinteberg. A parallel nonuniform fast fourier transform library based on an "exponential of semicircle" kernel. *SIAM Journal on Scientific Computing*, 41(5):C479–C504, 2019.

[2] Leslie Greengard and June-Yub Lee. Accelerating the nonuniform fast fourier transform. *SIAM Review*, 46(3):443–454, 2004.

[3] Dag Lindbo and Anna-Karin Tornberg. Spectral accuracy in fast Ewald-based methods for particle simulations. *Journal of Computational Physics*, 230(24):8744–8761, 2011.

Email address: erik.bostrom@mdu.se