A Dual-Space Multilevel Kernel-Splitting Algorithm for the Open Poisson Equation

MASTER THESIS

in Computational Science and Engineering

Eidgenössische Technische Hochschule (ETH) Zürich

written by

RYAN AMMANN

supervised by

Dr. A. Adelmann

advisors

S. Mayani

Dr. S. Muralikrishnan

Dr. M. Frey

May 2024

Abstract

This work implements a dual-space multilevel kernel-splitting approach to solve the open-boundaries Poisson equation for discrete sources and targets. It uses an octree structure to perform an adaptive domain decomposition based on the distribution of points. Similarly to Ewald summation, the solution is calculated by splitting the Laplace operator's Green's function into far-reaching and short-range constituents. The far-reaching interaction is computed at the octree's coarsest level and diagonalized by a Fourier transform. This is followed by successive correction terms that are calculated at every level. Finally, the short-range interaction is entirely local and is done at the leaf level. The octree and the solver are implemented in C++ within the Independent Parallel Particle Layer library, which functions as a performance-portable library for particle-mesh methods. The implementation uses the FINUFFT C++ library to compute non-uniform fast Fourier transforms. The performance is benchmarked against the explicit convolution of the Green's function with the sources in serial and with 10 cores over a single MPI rank.

Contents

Chapter 1

Introduction

The Poisson equation is one of the most fundamental partial differential equations (PDEs). It is crucial in many scientific and engineering applications, such as electrostatics, fluid dynamics, and heat transfer. Solving the Poisson equation efficiently, especially in the context of large-scale particle simulations, is typically done with one of two classes of algorithms.

The first is the class of uniform grid-based methods [7]. This approach splits the interaction into a long-range and short-range component. The far-reaching interaction term is efficiently diagonalized by an FFT since it has compact support in Fourier space. The short-range terms decay exponentially and only need to be calculated between particles within neighboring cells. However, uniform grid-based methods face challenges in long simulations where the point distribution can become non-uniform. This non-uniformity can lead to an unbounded cost of the short-range terms as the number of points within each grid cell grows, making this approach inefficient.

The second is the class of adaptive tree-based methods, such as fast multipole methods (FMM) [5] [4]. A tree structure is used to decompose the domain based on the distribution of points. It is constructed by continuously refining leaf nodes that contain a large number of points. For particles within a cell, their collective effect on distant particles can be approximated using multipole expansions. Similarly to the uniform grid-based methods, the local interactions are only performed between neighboring leaf cells.

This work implements a dual-space multilevel kernel-splitting (DMK) method described in Jiang et al.[6]. It combines the tree-based domain decomposition of fast multipole methods with the kernel-splitting approach of uniform grid-based methods. The long-range term is approximated by a short Fourier transform over the entire domain. The residual contribution is done explicitly and only between leaf nodes. For every level between each leaf node and the root node, a localized correction term needs to be calculated.

Chapter 2

Mathematical Foundation

2.1 Fourier Transform

For this work, we only need to consider the Fourier transforms of functions belonging to the set of Schwartz functions $\mathcal{S}$ [11].

Definition 1. Schwartz functions are smooth functions on $\mathbb{R}^n$ with rapidly decreasing derivatives.

$$\mathcal{S} := \{f \in C^{\infty}(\mathbb{R}^n) \mid \forall \boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{N}^n, \ ||f||_{\boldsymbol{\alpha},\boldsymbol{\beta}} < \infty\} \tag{2.1}$$

where

$$||f||_{\boldsymbol{\alpha},\boldsymbol{\beta}} := \sup_{\boldsymbol{x}\in\mathbb{R}^n} |\boldsymbol{x}^{\boldsymbol{\alpha}}(\boldsymbol{D}^{\boldsymbol{\beta}} f)(\boldsymbol{x})|\,, \tag{2.2}$$

$\boldsymbol{x}^{\boldsymbol{\alpha}} = x_1^{\alpha_1} \cdots x_n^{\alpha_n}$, $\boldsymbol{D}^{\boldsymbol{\beta}} = \delta_1^{\beta_1} \cdots \delta_n^{\beta_n}$ and $\delta_i^{\beta_i} = (\frac{d}{dx})^{\beta_i}$.

Definition 2. The Fourier transform of a Schwartz function $f : \mathbb{R}^3 \to \mathbb{R}$ is defined as

$$\hat{f}(\boldsymbol{k}) = \mathcal{F}\{f\}(\boldsymbol{k}) = \int_{\mathbb{R}^3} f(\boldsymbol{x}) e^{-i\boldsymbol{k}\cdot\boldsymbol{x}}\, d\boldsymbol{x}, \quad \forall \boldsymbol{k} \in \mathbb{R}^3. \tag{2.3}$$

The inverse Fourier transform is defined as

$$f(\boldsymbol{x}) = \mathcal{F}^{-1}\{\hat{f}\}(\boldsymbol{x}) = \frac{1}{(2\pi)^3} \int_{\mathbb{R}^3} \hat{f}(\boldsymbol{k}) e^{i\boldsymbol{k}\cdot\boldsymbol{x}}\, d\boldsymbol{k}, \quad \forall \boldsymbol{x} \in \mathbb{R}^3. \tag{2.4}$$

Property 1. A useful property of the Fourier transform is dilation.

$$\mathcal{F}\{f(x/c)\}(\boldsymbol{k}) = |c|^d \hat{f}(|c|\boldsymbol{k}), \quad \text{for} \quad c \in \mathbb{R} \quad \text{with} \quad c \neq 0. \tag{2.5}$$

It describes how scaling the input to a function affects its Fourier transform. The famous uncertainty principle directly follows from this property. The more localized a function gets, the more spaced out its Fourier transform becomes.

2.2 Convolutional Theorem

Theorem 1. For two Schwartz functions f and g the Fourier transform of the convolution of the two functions is equivalent to the point-wise multiplication of the Fourier transforms of the two functions

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}. \tag{2.6}$$

Proof.

$$\begin{aligned}
\mathcal{F}\{f * g\}(\mathbf{k}) &= \int_{\mathbb{R}^n} (f * g)(\mathbf{x}) e^{-i\mathbf{k}\cdot\mathbf{x}}\, d\mathbf{x} && (2.7)\\
&= \int_{\mathbb{R}^n} \left(\int_{\mathbb{R}^n} f(\mathbf{y}) g(\mathbf{x}-\mathbf{y})\, d\mathbf{y} \right) e^{-i\mathbf{k}\cdot\mathbf{x}}\, d\mathbf{x} && (2.8)\\
&= \int_{\mathbb{R}^n} f(\mathbf{y}) \left(\int_{\mathbb{R}^n} g(\mathbf{x}-\mathbf{y}) e^{-i\mathbf{k}\cdot\mathbf{x}}\, d\mathbf{x} \right) d\mathbf{y}, && (2.9)
\end{aligned}$$

substituting $\mathbf{u} = \mathbf{x} - \mathbf{y}$ we get

$$\begin{aligned}
\mathcal{F}\{f * g\}(\mathbf{k}) &= \int_{\mathbb{R}^n} f(\mathbf{y}) \left(\int_{\mathbb{R}^n} g(\mathbf{u}) e^{-i\mathbf{k}\cdot(\mathbf{u}+\mathbf{y})}\, d\mathbf{u} \right) d\mathbf{y} && (2.10)\\
&= \int_{\mathbb{R}^n} f(\mathbf{y}) e^{-i\mathbf{k}\cdot\mathbf{y}} \left(\int_{\mathbb{R}^n} g(\mathbf{u}) e^{-i\mathbf{k}\cdot\mathbf{u}}\, d\mathbf{u} \right) d\mathbf{y} && (2.11)\\
&= \left(\int_{\mathbb{R}^n} g(\mathbf{u}) e^{-i\mathbf{k}\cdot\mathbf{u}}\, d\mathbf{u} \right) \left(\int_{\mathbb{R}^n} f(\mathbf{y}) e^{-i\mathbf{k}\cdot\mathbf{y}}\, d\mathbf{y} \right) && (2.12)\\
&= \hat{g}(\mathbf{k}) \cdot \hat{f}(\mathbf{k}). && (2.13)
\end{aligned}$$

□

2.3 Green's Function

Definition 3. Given a linear operator $\mathcal{L}$ acting on a set of functions. The Green's function of $\mathcal{L}$ satisfies

$$\mathcal{L}G(\boldsymbol{x} - \boldsymbol{y}) = \delta(\boldsymbol{x} - \boldsymbol{y}), \tag{2.14}$$

where δ is the Dirac delta function.

Property 2. Given a differential operator $\mathcal{L}$ with Green's function G and the partial differential equation

$$\mathcal{L}u(\boldsymbol{x}) = \rho(\boldsymbol{x}), \tag{2.15}$$

the solution is given by

$$u(\boldsymbol{x}) = (G * f)(\boldsymbol{x}). \tag{2.16}$$

Proof.

$$\begin{aligned}
\mathcal{L}(G * f)(\boldsymbol{x}) &= \mathcal{L} \int_{\mathbb{R}^n} G(\boldsymbol{x} - \boldsymbol{s}) f(\boldsymbol{x}) d\boldsymbol{s} && (2.17)\\
&= \int_{\mathbb{R}^n} \mathcal{L}G(\boldsymbol{x} - \boldsymbol{s}) f(\boldsymbol{x}) d\boldsymbol{s} && (2.18)\\
&= \int_{\mathbb{R}^n} \delta(\boldsymbol{x} - \boldsymbol{s}) f(\boldsymbol{x}) d\boldsymbol{s} && (2.19)\\
&= f(\boldsymbol{x}) && (2.20)
\end{aligned}$$

□

The Greens function of the 3D Laplace operator Δ is given by

$$G_\Delta(\boldsymbol{x} - \boldsymbol{y}) = \frac{1}{|\boldsymbol{x} - \boldsymbol{y}|}. \tag{2.21}$$

Chapter 3

Octree

An octree is a hierarchical tree structure where each internal node branches into eight child nodes. In this context, it facilitates an adaptive multi-level decomposition of a three-dimensional domain. Each node corresponds to a cube subset of the domain, with leaf nodes representing regions without further subdivisions. The tree is constructed such that there is a large number of leaf nodes where the concentration of points is high and few leaf nodes where points are scarce.

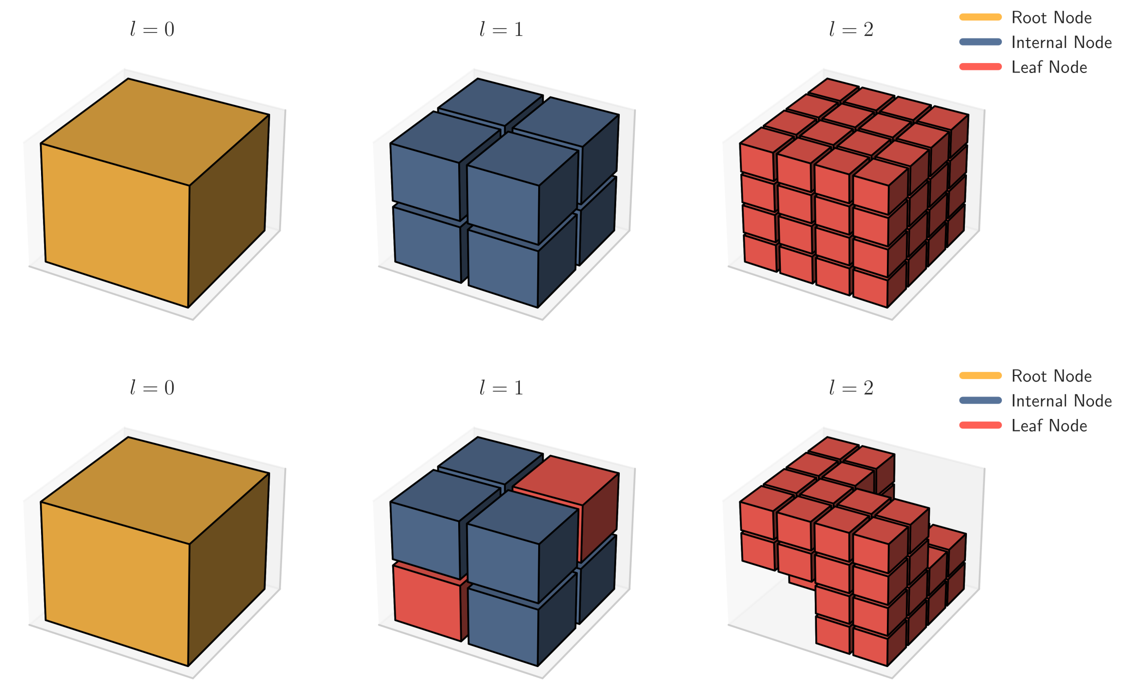

Figure 3.1: The top plot shows an octree of depth 2 for a uniform distribution of points and the resulting uniform domain decomposition. The root node (yellow) at level $l = 0$ covers the entire domain. It has eight children nodes (grey) at level $l = 1$. Each node at level $l = 1$ has eight child nodes, resulting in the 64 leaf nodes at level $l = 2$. The lower plot illustrates an octree structure, wherein leaf nodes are situated at different levels due to varying point densities. Specifically, the top-right and bottom-left corners exhibit lower point concentrations, resulting in leaf nodes positioned at level $l = 1$ for those regions.

Some important tree relations:

- A node B is called an **internal node** if it is not a leaf node
- The set of nodes that share a boundary point with node B and have the same depth l_B are called **colleagues** $\mathcal{C}(B)$
- The set of nodes that share a boundary point with node B and have depth $l_B - 1$ are called **coarse neighbors** $\mathcal{N}(B)$

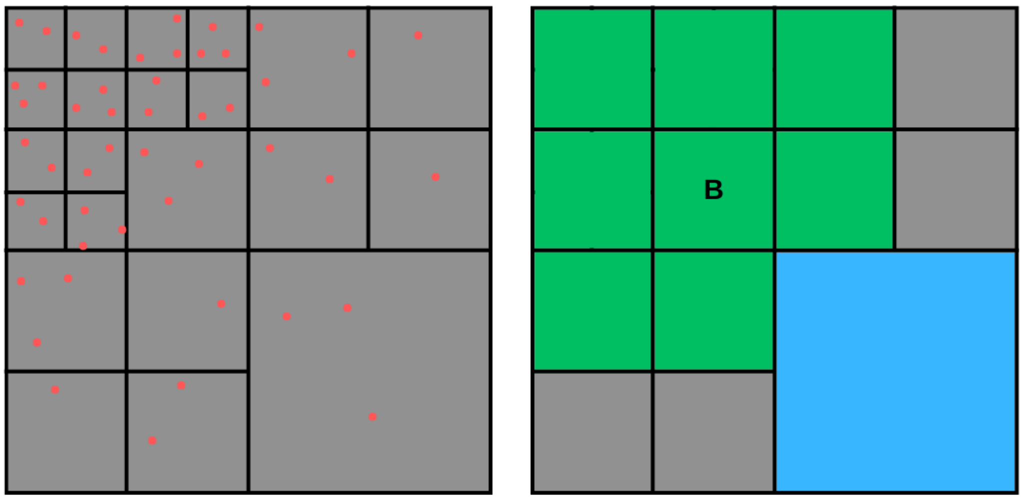

Figure 3.2: On the left, a representation of leaf nodes of a tree (quadtree). There are fine (deep) leaf nodes where the concentration of points is high and coarse leaf nodes where the concentration of points is low. On the right, the node B has the set of colleagues $\mathcal{C}(B)$ marked in green and the set of coarse neighbors $\mathcal{N}(B)$ marked in blue.

3.1 Representation

For this work, a linear data structure is used to save the nodes of the tree. In particular, each node is assigned a unique key and saved inside a map. The advantage of this kind of representation over a traditional pointer-based representation is that any given node can be accessed without having first to traverse through its parent. The encoding used for this is Morton Z-type encoding [14], also known as Z-order curve encoding. It is a way of mapping multidimensional gridpoints into one dimension while preserving the information about the location of the gridpoints. A node's key is encoded by interleaving the binary bits of its coordinate on the grid of nodes. To ensure that nodes at different levels of the tree have distinct keys, a level-dependent offset $c_l = 2^{d \cdot l}$, where d is the dimension and l is the depth, is added to each node's Morton key.

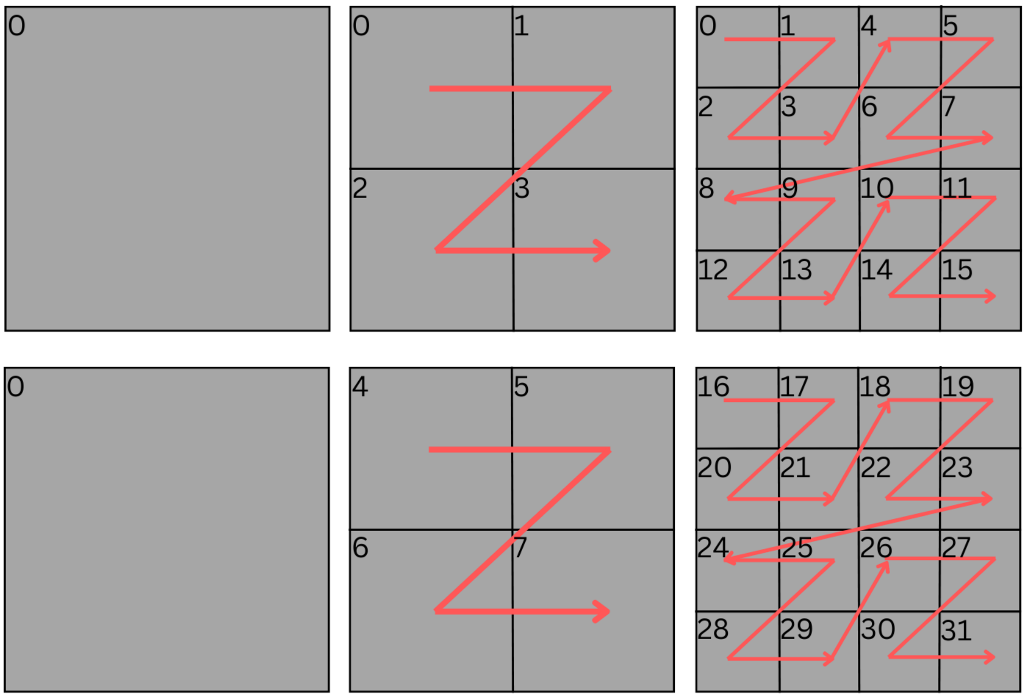

Figure 3.3: On top, the Morton encoding of the first 3 levels of a quadtree. Each gridpoint is ordered by the characteristic "Z-curve". On the bottom, the level-dependent offset $c_l = 2^{d \cdot l}$ is added, giving each node a unique key.

Example 1. A node B with grid coordinates $(x, y, z) = (1, 2, 3)$ has Morton encoding

$$\boldsymbol{M}(1, 2, 3) = \boldsymbol{M}(001, 010, 011) = 000110101 = 53 \tag{3.1}$$

3.2 Construction

The construction of the octree is characterized by two parameters that need to be defined prior. The first parameter is L_{max}, it specifies the maximally allowed depth of the tree. The root node is a level $l = 0$ and the deepest leaf node can sit at $l = L_{max}$. The second parameter is n_s, which is the maximal number of points allowed per leaf node. If a node contains more than n_s points, it is subdivided into eight children nodes.

The construction starts by creating a root node that covers the entire domain. If the number of points in the root node is less than or equal to n_s, the algorithm terminates since the root node itself is a valid leaf node. If this is not the case, eight children nodes are created, and the points belonging to each child are determined. The process repeats for each child node until a valid leaf node is reached. The implementation details are described in section 5.2.

Algorithm 1 Recursive function for top-down tree construction

```
function ADDNODES(node $\mathcal{B}$, points $\mathcal{P}$)
    if Number of points in $\mathcal{P} < n_s$ or Depth is $L_{max}$ then
        Fill points $\mathcal{P}$ into node $\mathcal{B}$
    else
        Initialize set of child nodes $\mathcal{D}(B)$
        for each child node $D \in \mathcal{D}(B)$ do
            Determine subset of points $\mathcal{P}_C \subset \mathcal{P}$ belonging to child $C$
            ADDNODES$(C, \mathcal{P}_C)$
        end for
    end if
end function
```

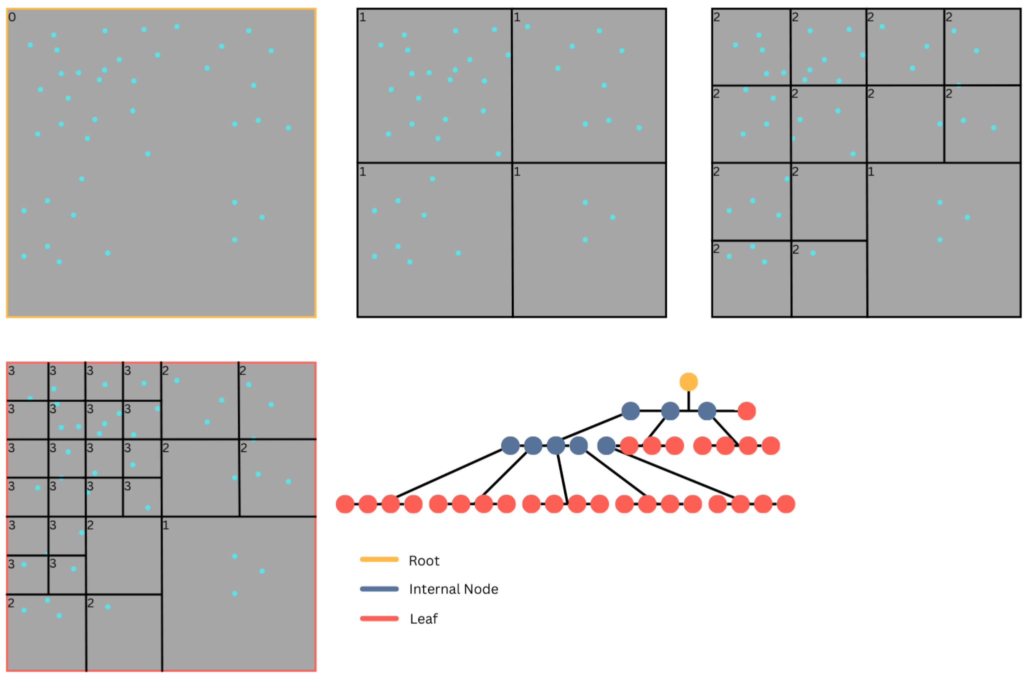

Figure 3.4: Illustration of the construction of a quadtree with parameter $n_s = 3$.

3.3 Balancing

In many applications of adaptive trees, a smoothness condition needs to be fulfilled. For this work, leaf nodes that share a boundary point must be no more than one level apart in depth. While there are octree construction algorithms that guarantee this balance condition [12], the one used here does not. Hence, the tree needs to be separately balanced after construction. Parallel tree balancing is possible and necessary for applications of dynamically changing octrees, such as in large fluid-dynamics simulations, but since this work does not have a time component, a serial algorithm is implemented. The balancing commences by sorting all leaves inside a queue in descending Morton order, which means the deepest leaf nodes are at the front of the queue. The keys of all the possible colleagues of the current node are gathered. Then, the existence of each of these nodes is checked. For colleagues that do not exist, the depth of the next existing ancestor is compared to the current node's depth. For ancestors that are more than one level higher in depth are refined and their children nodes are appended to the back of the queue. This process is repeated for the entire queue until it is empty. The implementation details are described in section 5.2.

Algorithm 2 Balancing of the Octree

```
Require: Sorted queue of leaves $\mathcal{Q}$, deepest leaves at the front
Ensure: Balanced Octree
  while $\mathcal{Q}$ is not empty do
      $l \leftarrow \mathcal{Q}.front$
      IsBalanced $\leftarrow true$
      Determine all possible colleagues $\mathcal{C}(l)$
      for all Colleagues $c \in \mathcal{C}(l)$ that don't exist, find next existing ancestor $\mathcal{A}(c)$ do
          if The difference in depth between $\mathcal{A}(c)$ and $l$ is more than 1 then
              Refine node $\mathcal{A}(c)$
              Push new child nodes of $\mathcal{A}(c)$ to the back of $\mathcal{Q}$
              IsBalanced $\leftarrow false$
          end if
      end for
      if IsBalanced $== true$ then
          Pop $l$ from the queue $\mathcal{Q}$
      end if
  end while
```

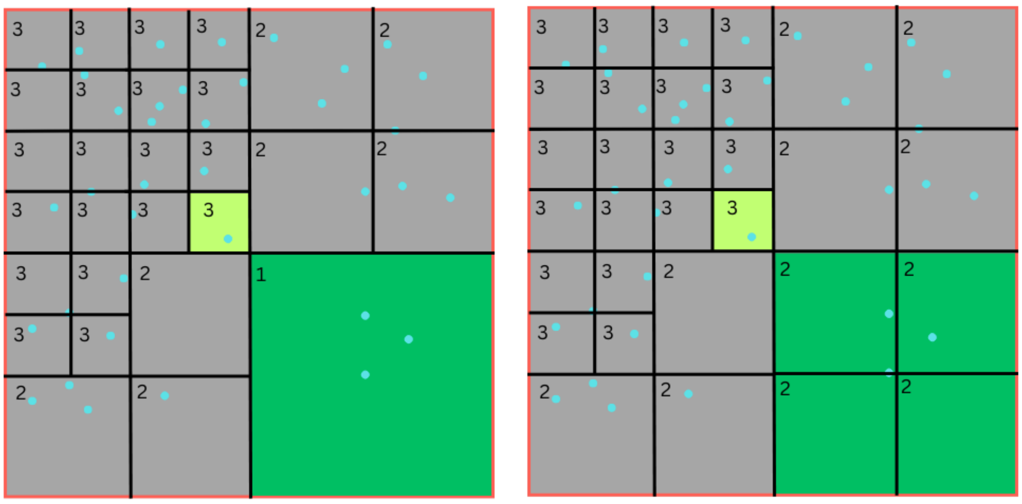

Figure 3.5: On the left, the light green leaf node has depth $l = 3$ and shares a boundary point with a leaf node at level $l = 1$ (dark green), making this tree unbalanced. On the right, the coarse leaf node has been refined, making the difference in level $\Delta l = 1$ and the tree balanced.

Chapter 4

DMK Algorithm

This section describes the details of the algorithm first presented in [6].
Given a set of source points $S = \{\boldsymbol{x}_j\}_{j=1}^N$ with charges $\{\rho_j\}_{j=1}^N$ and a set of target points $T = \{\boldsymbol{x}_i\}_{i=1}^N$ on a domain $\Omega \subset \mathbb{R}^3$. We want to find the solution to the Poisson equation at the target points $\{u(\boldsymbol{x}_i)\}$. In the three-dimensional case, the Green's function for the Laplace operator is

$$G_\Delta(\boldsymbol{x}_i, \boldsymbol{x}_j) = \frac{1}{|\boldsymbol{x}_i - \boldsymbol{x}_j|} = \frac{1}{r_{ij}} = G(r_{ij}). \tag{4.1}$$

The solution can be explicitly written as

$$u(\boldsymbol{x}_i) = \sum_j G(r_{ij})\rho_j, \quad \text{where} \quad \boldsymbol{x}_i \in T, \boldsymbol{x}_j \in S. \tag{4.2}$$

4.1 General Kernel Splitting

The Green's function kernel can be broken up into a far-reaching term $M(r)$ and a local term $R(r)$

$$G(r) = \frac{\operatorname{erf}(r/\sigma)}{r} + \frac{\operatorname{erfc}(r/\sigma)}{r} =: M(r) + R(r), \tag{4.3}$$

where $\operatorname{erf}(r)$ and $\operatorname{erfc}(r)$ are the error function and complementary error function, respectively.

$$\operatorname{erf}(r) = \frac{2}{\sqrt{\pi}} \int_0^r e^{-t^2} dt \quad \text{and} \quad \operatorname{erfc}(r) = 1 - \operatorname{erf}(r)\,. \tag{4.4}$$

As illustrated in the figure 4.1, the parameter σ controls the shape of the kernel decomposition. In the algorithm, this parameter is chosen such that the local term decays within the diameter of the leaf node. The convolution in equation 4.2 can be rewritten as

$$u(\boldsymbol{x}_i) = \sum_j M(r_{ij})\rho_j + \sum_j R(r_{ij})\rho_j\,. \tag{4.5}$$

This is the approach typically used in uniform grid-based methods. The far-reaching part $M(r)$ behaves like $1/r$ and needs to be calculated between all sources and targets. However, this can be sped by performing the convolution in Fourier space, where it becomes a point-wise multiplication, going from $O(N^2)$ to $O(N \log N)$. The residual term decays exponentially and only needs to be calculated between close points. The parameter σ is chosen such that $R(r)$ is compactly supported (up to precision ϵ) within the length of a grid cell. This means that the residual contribution only needs to be calculated for points in the set of colleagues of a cell. This approach is good for point distributions that are relatively uniform. However, the limitation of having a fixed cell resolution means that for non-uniform distribution of points, the effort of calculating the short-range terms can become unbounded.

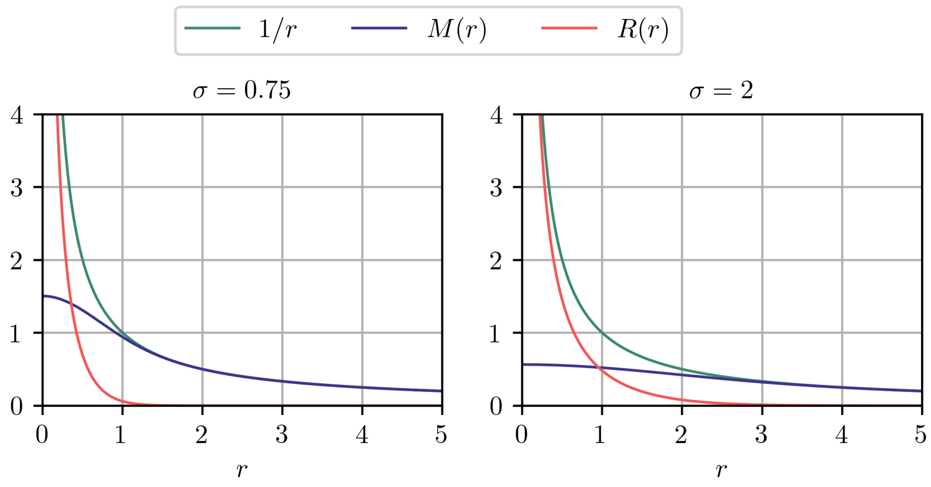

Figure 4.1: Decomposition of the Green's function with two different values of σ

4.2 Level-wise Kernel Splitting

The octree produces a multi-level, hierarchical set of meshes that become finer where the concentration of points is high and stay coarse where points are scarce. Consequently, there is a need for a multi-level decomposition of the Laplace kernel. Consider the telescoping decomposition of the Laplace kernel for each level $L \in \{0, ..., L_{max}\}$.

$$\frac{1}{r} = M_0(r) + \sum_{l=0}^{L-1} D_l(r) + R_L(r), \quad L = 0, ..., L_{max}, \tag{4.6}$$

where

$$M_0(r) = \frac{\text{erf}(r/\sigma_0)}{r}, \qquad \textit{(far-field)} \tag{4.7}$$

$$D_l(r) = \frac{\text{erf}(r/\sigma_{l+1}) - \text{erf}(r/\sigma_l)}{r}, \qquad \textit{(difference correction)} \tag{4.8}$$

$$R_L(r) = \frac{\text{erfc}(r/\sigma_L)}{r}, \qquad \textit{(residual)} \tag{4.9}$$

for $\sigma_0 > \sigma_1 > ... > \sigma_{L_{max}}$. It is a valid decomposition of the kernel at each level L. The term $R_L(r)$ is only evaluated for leaf nodes at level L. The values σ_l are chosen so that the residual kernel is supported within nearest neighbors at that same level l. This is accomplished by setting

$$\sigma_0 \approx \frac{r_0}{\sqrt{\log(1/\epsilon)}}, \quad \sigma_l = \sigma_{l/2}/2. \tag{4.10}$$

$M_0(r)$ is still the far-field contribution calculated on the coarsest level (over the entire domain). The terms $D_l(r)$ are the necessary correction terms for using different values of σ for the far-field and residual terms. The corrected far-field term for a leaf node at level l can be written as

$$M_L(r) = M_0(r) + \sum_{l=0}^{L-1} D_l(r). \tag{4.11}$$

4.3 Far-field Contribution

The far-field contribution is the convolution of the sources with the far-reaching part of the Green's function kernel

$$u_{\text{far}}(\boldsymbol{x}_i) = \sum_j M(r_{ij})\rho_j = \sum_j \frac{\text{erf}(r_{ij}/\sigma)}{r_{ij}}\rho_j\,. \tag{4.12}$$

Calculating the far-field kernel in real space requires $O(N^2)$ work. This calculation is amenable to a Fourier-based method, which can be done in $O(N\log N)$. By the Convolutional theorem, the far-field contribution from equation 4.12 is equal to

$$u_{\text{far}}(\boldsymbol{x}_i) = \mathcal{F}^{-1}(\boldsymbol{x}_i)\{\hat{M} * \hat{\rho}\} = \frac{1}{(2\pi)^3}\int_{\boldsymbol{k}\in\mathbb{R}^3} e^{i\boldsymbol{k}\cdot\boldsymbol{x}_i}\hat{M}(\boldsymbol{k})\hat{\rho}(\boldsymbol{k})d\boldsymbol{k}, \tag{4.13}$$

$$\text{where}\quad \hat{M}(\boldsymbol{k}) = \int_\Omega e^{-i\boldsymbol{k}\cdot\boldsymbol{x}}M(\boldsymbol{x})d\boldsymbol{x} = 4\pi\frac{e^{-\boldsymbol{k}^2\sigma^2/4}}{\boldsymbol{k}^2} \tag{4.14}$$

$$\text{and}\quad \hat{\rho}(\boldsymbol{k}) = \sum_j e^{-i\boldsymbol{k}\cdot\boldsymbol{x}_j}\rho_j. \tag{4.15}$$

The Fourier transformed far-field kernel $\hat{M}(\boldsymbol{k})$ decays exponentially but is singular around $\boldsymbol{k} = \boldsymbol{0}$. This makes the numerical evaluation of equation 4.13 expensive. A workaround is to replace $M(\boldsymbol{x})$ with a "windowed kernel" $W(\boldsymbol{x})$ that has non-singular Fourier transform $\widehat{W}(\boldsymbol{k})$. The basic idea is that the replacement $W(\boldsymbol{x})$ closely follows the true kernel inside the simulation domain and only differs outside. Given the diameter of the domain $C = \sqrt{3}$ we chose the windowed kernel

$$W(r) = \frac{\text{erf}(r/\sigma)}{r} - \frac{1}{2}\left(\frac{\text{erf}((b\sigma + C + r)/\sigma)}{r} - \frac{\text{erf}((b\sigma + C - r)/\sigma)}{r}\right). \tag{4.16}$$

The approximation error for using $W(r)$ is given by

$$|W(r) - M(r)| = \frac{\text{erf}((b\sigma + C + r)/\sigma) - \text{erf}((b\sigma + C - r)/\sigma)}{2r} \tag{4.17}$$

$$\leq \frac{\text{erfc}(b)}{r}, \quad r \leq C. \tag{4.18}$$

The error can be controlled by the parameter b. For example, setting $b = 6$ yields 15 digits of accuracy.

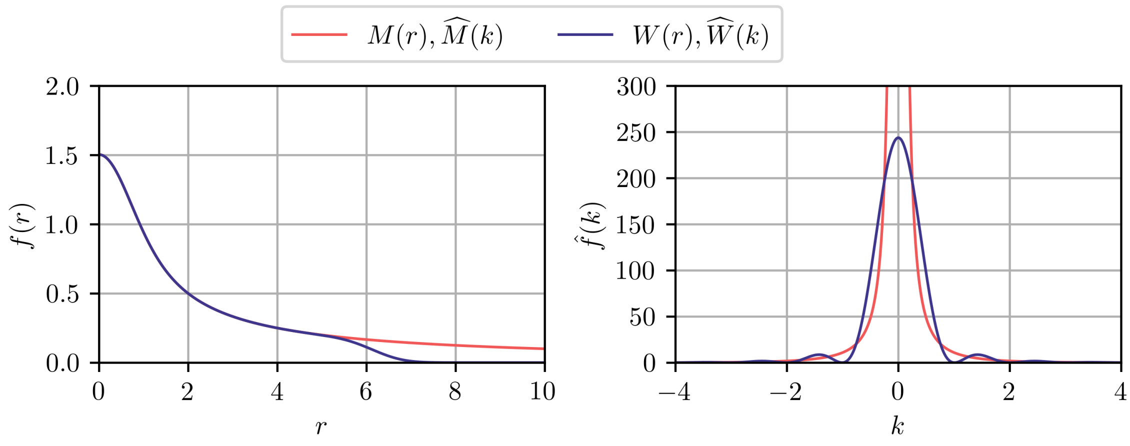

Figure 4.2: Comparison of the windowed kernel (red) with the original far-field kernel (blue).

This kernel has smooth and, importantly, non-singular Fourier transform

$$\widehat{W}(\boldsymbol{k}) = 8\pi \left(\frac{\sin(\tilde{C}|\boldsymbol{k}|/2)}{|\boldsymbol{k}|} \right)^2 e^{-\boldsymbol{k}^2\sigma^2/4} \quad \text{with} \quad \tilde{C} = C + b\sigma. \tag{4.19}$$

Substituting in $\widehat{M}(\boldsymbol{k})$ with $\widehat{W}(\boldsymbol{k})$ in equation 4.13 gives

$$u_{\text{far}}(\boldsymbol{x}_i) \approx \frac{1}{\pi^2} \int_{k\in\mathbb{R}^3} e^{i\boldsymbol{k}\cdot\boldsymbol{x}_i} \left(\frac{\sin(\tilde{C}|\boldsymbol{k}|/2)}{|\boldsymbol{k}|} \right)^2 e^{-\boldsymbol{k}^2\sigma^2/4} \hat{\rho}(\boldsymbol{k})\, d\boldsymbol{k}. \tag{4.20}$$

To numerically compute this term, the integral over the modes $\boldsymbol{k} \in \mathbb{R}$ is replaced with a sum over discrete points $\boldsymbol{k}$ in Fourier space. Theoretically, full knowledge of the integrand is required to choose suitable quadrature points. In practice, we do not have this information as $\hat{\rho}(\boldsymbol{k})$ depends on the number of points, as well as their distribution. We heuristically choose the relation

$$K_{max}^2 \geq 4\log(1/\epsilon)/\sigma^2 \tag{4.21}$$

for the integration boundaries. This choice is based on the behavior of $\widehat{W}(\boldsymbol{k})$. Plugging this inequality into equation 4.19, we see that the absolute value of $\widehat{W}(\boldsymbol{k})$ beyond K_{max} is far below ϵ. To determine the number of quadrature points, we look at the number of oscillations in the interval $[-K_{max}, K_{max}]$ of $\widehat{W}$ in each dimension. It is proportional to $K_{max}\tilde{C}/\pi \sim \sqrt{\log(1/\epsilon)}/\sigma$. By Nyquist's sampling theorem, the number of quadrature points should be at least twice the number of oscillations to avoid an aliasing error. In equation 4.10 of the previous section, σ_0 is chosen with the support of the residual kernel at the leaf level in mind. This choice of σ_0 is so large that

$$n_f \approx 4\log(1/\epsilon) \tag{4.22}$$

quadrature points per dimension is enough to achieve ϵ precision. The distance between grid points in Fourier space is denoted by $h_0 = 2K_{max}/n_f$. The equation 4.20 can now be rewritten

$$u_{\text{far}}(\boldsymbol{x}_i) \approx \frac{1}{\pi^2} \sum_{\boldsymbol{k}} e^{i\boldsymbol{k}\cdot\boldsymbol{x}_i} \left(\frac{\sin(\tilde{C}|\boldsymbol{k}|/2)}{|\boldsymbol{k}|} \right)^2 e^{-\boldsymbol{k}^2\sigma^2/4} \hat{\rho}(\boldsymbol{k}) \tag{4.23}$$

and $\hat{\rho}(\boldsymbol{k}) = \sum_j e^{-i\boldsymbol{k}\cdot\boldsymbol{x}_j}\rho_j$ is the Fourier transform of the sources onto the chosen modes, and $\boldsymbol{k} \in [-h_0 n_f, ..., h_0 n_f]^3$.

4.4 Level-wise Difference Correction

Recalling equations 4.6 and 4.8, the difference kernel is a level-wise correction for using different values of σ in the far-field and residual part of the calculation. For a target point $\boldsymbol{x}_i$ in a leaf node at level L the difference contribution is written as

$$u_{\text{diff}}(\boldsymbol{x}_i) = \sum_{l=0}^{L-1} u_{\text{diff},l}(\boldsymbol{x}_i) \tag{4.24}$$

and

$$u_{\text{diff},l}(\boldsymbol{x}_i) = \sum_j D_l(r_{ij})\rho_j. \tag{4.25}$$

For a target point in a node B at level l the contribution of the difference kernel D_l is negligible beyond the colleagues $\mathcal{C}(B)$ of B. As in the case of the far-field contribution, this convolution is more efficiently done in Fourier space. The Fourier-transformed difference kernel at level l is given by

$$\widehat{D}_l(\boldsymbol{k}) = \begin{cases} 4\pi(e^{-|\boldsymbol{k}|^2\sigma_{l+1}^2/4} - e^{-|\boldsymbol{k}|^2\sigma_l^2/4})/|k|^2, & \boldsymbol{k} \neq (0,0,0) \\ \pi(\sigma_l^2 - \sigma_{l+1}^2), & \boldsymbol{k} = (0,0,0)\,. \end{cases} \tag{4.26}$$

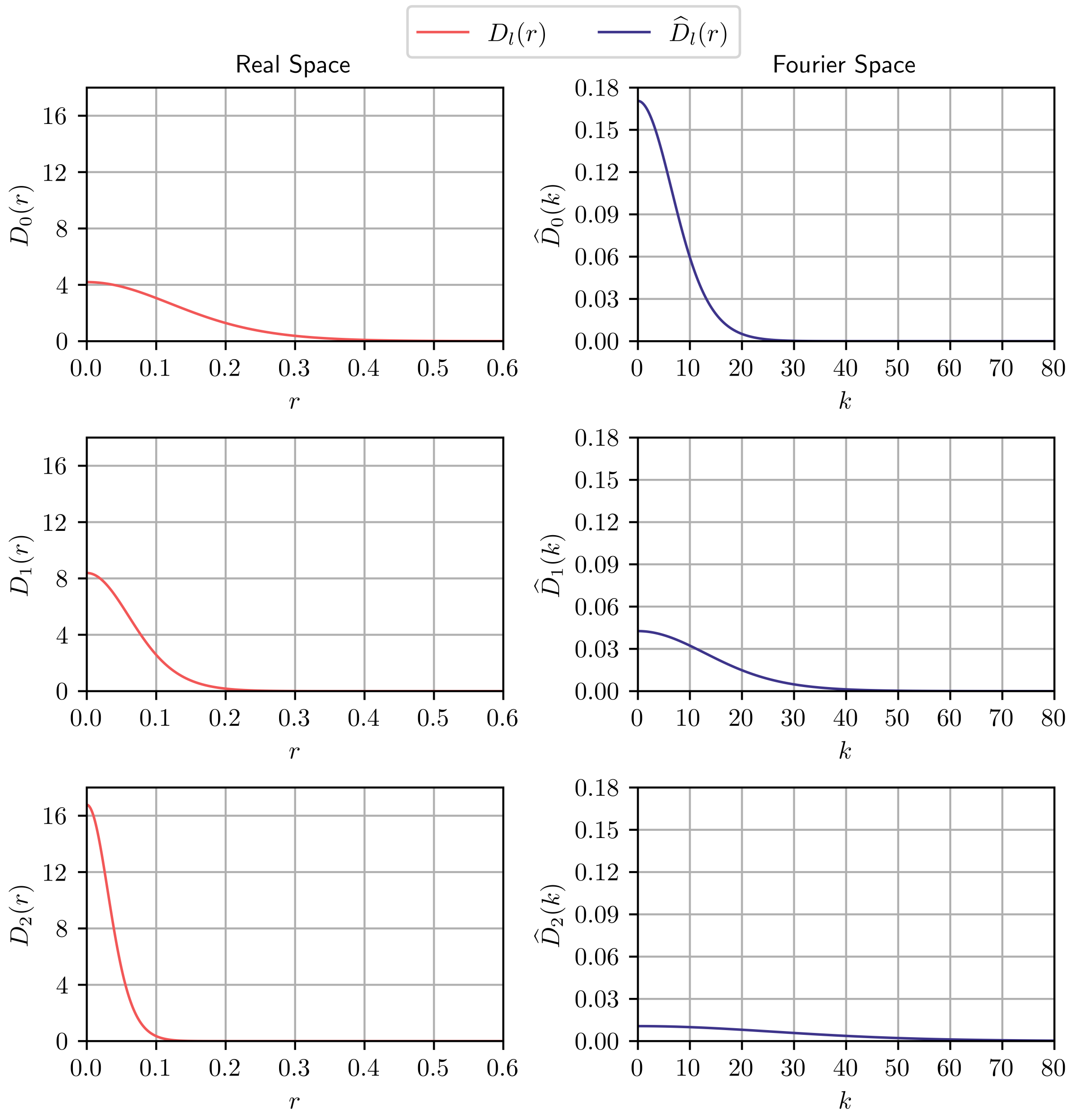

Figure 4.3: Differnce kernels in real space and Fourier space at levels $l = 0, 1, 2$. The support of the kernels in real space decreases with depth. Inversely, the support in Fourier space grows. As a result, the integration boundary increases with the depth of the node.

Let us now write the contribution of the difference correction at level l for a point $\boldsymbol{x}_i$ in node B. We denote the center of a node B as $\boldsymbol{c(B)}$.

$$
\begin{aligned}
u_{\text{diff},l}(\boldsymbol{x}_i) &= \sum_{\boldsymbol{y}_j} D_l(r_{ij})\rho_j && (4.27)\\
&= \sum_{S\in\mathcal{C}(B)} \sum_{\boldsymbol{y}_j\in S} D_l(r_{ij})\rho_j && (4.28)\\
&= \sum_{S\in\mathcal{C}(B)} \int_{\boldsymbol{k}\in\mathbb{R}^3} e^{i\boldsymbol{k}\cdot\boldsymbol{x}_i} \frac{1}{(2\pi)^3}\widehat{D}_l(\boldsymbol{k}) \sum_{\boldsymbol{y}_j\in S} e^{-i\boldsymbol{k}\cdot\boldsymbol{y}_j}\rho_j \, d\boldsymbol{k} && (4.29)\\
&= \int_{\boldsymbol{k}\in\mathbb{R}^3} e^{i\boldsymbol{k}\cdot\boldsymbol{x}_i} \sum_{S\in\mathcal{C}(B)} \frac{1}{(2\pi)^3}\widehat{D}_l(\boldsymbol{k}) \sum_{\boldsymbol{y}_j\in S} e^{-i\boldsymbol{k}\cdot\boldsymbol{y}_j}\rho_j \, d\boldsymbol{k} && (4.30)\\
&= \int_{\boldsymbol{k}\in\mathbb{R}^3} e^{i\boldsymbol{k}\cdot\boldsymbol{x}_i} e^{-i\boldsymbol{k}\cdot\boldsymbol{c(B)}} \sum_{S\in\mathcal{C}(B)} e^{i\boldsymbol{k}\cdot\boldsymbol{c(B)}} \frac{1}{(2\pi)^3}\widehat{D}_l(\boldsymbol{k}) \sum_{\boldsymbol{y}_j\in S} e^{-i\boldsymbol{k}\cdot\boldsymbol{y}_j}\rho_j \, d\boldsymbol{k} && (4.31)\\
&= \int_{\boldsymbol{k}\in\mathbb{R}^3} e^{i\boldsymbol{k}\cdot\boldsymbol{x}_i} e^{-i\boldsymbol{k}\cdot\boldsymbol{c(B)}} \sum_{S\in\mathcal{C}(B)} e^{i\boldsymbol{k}\cdot\boldsymbol{c(B)}} e^{-i\boldsymbol{k}\cdot\boldsymbol{c(S)}} \frac{1}{(2\pi)^3}\widehat{D}_l(\boldsymbol{k}) \sum_{\boldsymbol{y}_j\in S} e^{i\boldsymbol{k}\cdot\boldsymbol{c(S)}} e^{-i\boldsymbol{k}\cdot\boldsymbol{y}_j}\rho_j \, d\boldsymbol{k} && (4.32)\\
&= \int_{\boldsymbol{k}\in\mathbb{R}^3} e^{i\boldsymbol{k}\cdot(\boldsymbol{x}_i-\boldsymbol{c(B)})} \underbrace{\sum_{S\in\mathcal{C}(B)} e^{i\boldsymbol{k}\cdot(\boldsymbol{c(B)}-\boldsymbol{c(S)})} \frac{1}{(2\pi)^3}\widehat{D}_l(\boldsymbol{k}) \underbrace{\sum_{\boldsymbol{y}_j\in S} e^{i\boldsymbol{k}\cdot(\boldsymbol{c(S)}-\boldsymbol{y}_j)}\rho_j}_{\Phi_l[S]}}_{\Psi_l[B]} \, d\boldsymbol{k} && (4.33)
\end{aligned}
$$

- (4.27) is the definition of the level l difference correction
- (4.28) uses that the D_l is negligible beyond the colleagues $\mathcal{C}(B)$
- (4.29) uses the Convolutional theorem
- (4.30) uses linearity to switch the integration and the sum
- (4.31) multiplies the equation with $e^{i\boldsymbol{k}\cdot\boldsymbol{c(B)}}e^{-i\boldsymbol{k}\cdot\boldsymbol{c(B)}}$ in order express target positions relative to the node center
- (4.32) multiplies the equation with $e^{i\boldsymbol{k}\cdot\boldsymbol{c(B)}}e^{-i\boldsymbol{k}\cdot\boldsymbol{c(S)}}$ in order express source positions relative to the colleague's node center
- In (4.33) we can identify the outgoing expansions $\Phi_l[S]$ for colleagues $S \in \mathcal{C}(B)$ and incoming expansion $\Psi_l[B]$ for node B

The outgoing expansion $\Phi_l[S]$ is defined as

$$\Phi_l[S](\boldsymbol{k}) = \sum_{\boldsymbol{y}_j\in S} e^{i\boldsymbol{k}\cdot(\boldsymbol{c(S)}-\boldsymbol{y_j})}\rho_j. \tag{4.34}$$

It represents the Fourier transform of sources in node S, where the positions are given relative to the center of the node $\boldsymbol{c(S)}$. The incoming expansion $\Psi_l[B](\boldsymbol{k})$ shifts the positions by the new center $\boldsymbol{c(B)}$ and multiplies the sum of the outgoing expansions of all its colleagues by the Fourier-space difference kernel

$$\Psi_l[B](\boldsymbol{k}) = \frac{1}{(2\pi)^3} \sum_{S\in\mathcal{C}(B)} e^{i\boldsymbol{k}\cdot(\boldsymbol{c(B)}-\boldsymbol{c(S)})}\widehat{D}_l(\boldsymbol{k})\Phi_l[S](\boldsymbol{k})\,. \tag{4.35}$$

In that sense, it can be understood as the Fourier transform of the difference contribution for targets in node B. To numerically evaluate the integral in equation 4.33 we need to determine suitable integration bounds and the number of points. When proceeding from level l to $l+1$ the value σ decreases by a factor of two. As shown in figure 4.3 the relevant support of $\widehat{D}$ and, as a result, the integration boundaries K_l double. The level-dependent boundary for achieving precision ϵ is approximately

$$K_l \approx \frac{4}{r_l} \log\left(\frac{1}{\epsilon}\right). \tag{4.36}$$

At the same time, the distance between relevant sources and targets also decreases by a factor of two. This means the number of oscillations, and therefore, the number of required quadrature nodes per dimension n_f in the range $[-K_l, K_l]$ remains constant across levels. Choosing

$$n_f = \frac{6}{\pi} \log\left(\frac{1}{\epsilon}\right) \tag{4.37}$$

with integration boundaries from equation 4.36 ensures sufficient precision ϵ. We define h_l as the spacing between integration points in Fourier space

$$h_l = \frac{2K_l}{n_f}. \tag{4.38}$$

The level l difference correction for target point $\boldsymbol{x}_i \in B$ is written as

$$\begin{aligned} u_{\text{diff},l}(\boldsymbol{x}_i) &\approx \sum_{\boldsymbol{k}} e^{i\boldsymbol{k}\cdot(\boldsymbol{x}_i-\boldsymbol{c(B)})} \sum_{S\in\mathcal{C}(B)} e^{i\boldsymbol{k}\cdot(\boldsymbol{c(B)}-\boldsymbol{c(S)})} \frac{1}{(2\pi)^3} \widehat{D}_l(\boldsymbol{k}) \sum_{\boldsymbol{y}_j\in S} e^{i\boldsymbol{k}\cdot(\boldsymbol{c(S)}-\boldsymbol{y}_j)} \rho_j && (4.39) \\ &= \sum_{\boldsymbol{k}} e^{i\boldsymbol{k}\cdot(\boldsymbol{x}_i-\boldsymbol{c(B)})} \Psi_l[B](\boldsymbol{k}), && (4.40) \end{aligned}$$

where $\boldsymbol{k} \in [-h_l n_f, ..., h_l n_f]^3$.

4.5 Residual Contribution

The residual contribution for a point $\boldsymbol{x}_i$ inside a leaf node B at level l is given by

$$u_{\text{res},l}(\boldsymbol{x}_i) = \sum_{\boldsymbol{y}_j} R_l(r_{ij})\rho_j. \tag{4.41}$$

In section 4.2 the values for σ were chosen such that the residual kernel is compactly supported within the colleagues $\mathcal{C}(B)$ of leaf node B, allowing us to write

$$u_{\text{res},l}(\boldsymbol{x}_i) = \sum_{S\in\mathcal{C}(B)} \sum_{\boldsymbol{y}_j\in S} R_l(r_{ij})\rho_j\,. \tag{4.42}$$

During the octree's construction, the parameter n_s determines the number of points in leaf nodes and directly controls the cost of the residual contribution. It is chosen such that the cost of the far-field and difference corrections are balanced with the residual computation.

4.6 Full Algorithm

The discrete DMK algorithm

Input: We are given N discrete targetpoints $\boldsymbol{x}_i,\;\; i = 1, ..., N$ and source points $\boldsymbol{y}_j,\;\; j = 1, ..., N$ with source values $\rho_j,\;\; j = 1, ..., N$ inside the 3D unit box $[0, 1]^3$. The algorithm is prescribed a precision ϵ, maximally allowed depth of the octree L_{max}, and maximal number of elements per leaf node n_s.

Output: The potentials at the target locations that solve the free-space Poisson equation.

Step 0: Initialization

```
1: Construct an octree with parameters $L_{max}$ and $n_s$
2: Balance the octree
```

Step 1: Far-field Contribution

```
1: Compute integration boundaries $K_{max}$
2: Compute the number of quadrature points in Fourier space $n_f \approx 4\log(1/\epsilon)$ in each dimension
3: Determine $h_0 = 2K_{max}/n_f$
4: Initialize the grid in Fourier space $Z_0 = [-h_0 n_f, ..., h_0 n_f]^3$
5: Perform type 1 NUFFT of the sources onto the grid
       $$\hat{\rho}(\boldsymbol{k}) = \sum_{j=1}^{N} e^{-i\boldsymbol{k}\cdot\boldsymbol{y}_j}\rho_j,\;\; \boldsymbol{k} \in Z_0$$
6: Perform the point-wise multiplication of the Fourier-transformed sources $\hat{\rho}(\boldsymbol{k})$ with the Fourier-space (windowed) far-field kernel $\widehat{W}(\boldsymbol{k})$
       $$\hat{u}_{\text{far}}(\boldsymbol{k}) = \widehat{W}(\boldsymbol{k}) * \hat{\rho}(\boldsymbol{k}),\;\; \boldsymbol{k} \in Z_0$$
7: Perform type 2 NUFFT onto the target points
       $$u_{\text{far}}(\boldsymbol{x}_i) = \frac{1}{\pi^2}\sum_{\boldsymbol{k}\in Z_0} e^{i\boldsymbol{k}\cdot\boldsymbol{x}_i}\hat{u}_{\text{far}}(\boldsymbol{k}),\;\; i = 1, ..., N$$
```

Step 2: Level-wise Difference Correction

```
Calculate the number of Fourier modes per dimension to use for the level-wise difference
correction $n_f \approx 6/\pi \log(1/\epsilon)$

for level $l = 0, ..., L_{max} - 1$ do
    Calculate integration boundary $K_l$
    Determine $h_l = 2K_l/n_f$
    Initialize grid $Z_l = [-h_l n_f, ..., h_l n_f]^3$
    for each node $B$ at level $l$ do
        if $B$ is an internal node (not a leaf node) then
            Calculate outgoing expansion using type 1 NUFFT
                $$\Phi_l[B](\boldsymbol{k}) = \sum_{\boldsymbol{y}_j \in B} e^{-i\boldsymbol{k}\cdot(\boldsymbol{y}_j - \boldsymbol{c}(\boldsymbol{S}))}\rho_j, \quad \boldsymbol{k} \in Z_l$$
        end if
    end for
    for each node $B$ at level $l$ do
        Calculate incoming expansion from its colleagues $S \in \mathcal{C}(B)$
            $$\Psi[B](\boldsymbol{k}) = \frac{1}{(2\pi)^3} \sum_{S \in \mathcal{C}(B)} e^{i\boldsymbol{k}\cdot(\boldsymbol{c}(\boldsymbol{B}) - \boldsymbol{c}(\boldsymbol{S}))}\widehat{D}_l(\boldsymbol{k})\Phi_l[S](\boldsymbol{k}), \quad \boldsymbol{k} \in Z_l$$
    end for
    for each node $B$ at level $l$ do
        if $B$ has an incoming expansion $\Psi_l[B](\boldsymbol{k})$ then
            Calculate the level $l$ correction term for targets $\boldsymbol{x}_i \in B$ using type 2 NUFFT
                $$u_{\text{diff},l}(\boldsymbol{x}_i) = \sum_{\boldsymbol{k} \in Z_l} e^{i\boldsymbol{k}\cdot(\boldsymbol{x}_i - \boldsymbol{c}(\boldsymbol{B}))}\Psi_l[B](\boldsymbol{k}), \quad \boldsymbol{x}_i \in B$$
        end if
    end for
end for
```

Step 3: Residual Contribution

For the residual contribution, each leaf node B adds the contribution of its own source points to the targets in all of its colleagues $\mathcal{C}(B)$ and coarse neighbors $\mathcal{N}(B)$.

```
for each leaf node $B$ do
    Determine level $l$ of $B$
    for each colleague $S \in \mathcal{S}(B)$ do
        $u_{\text{res}}(\boldsymbol{x}_i) \mathrel{+}= \sum_{\boldsymbol{y}_j \in B} R_l(|\boldsymbol{x}_i - \boldsymbol{y}_j|)\rho_j, \ \forall \boldsymbol{x}_i \in S$
    end for
    for each coarse neighbor $S \in \mathcal{N}(B)$ do
        $u_{\text{res}}(\boldsymbol{x}_i) \mathrel{+}= \sum_{\boldsymbol{y}_j \in B} R_l(|\boldsymbol{x}_i - \boldsymbol{y}_j|)\rho_j, \ \forall \boldsymbol{x}_i \in S$
    end for
end for
```

Runtime Analysis

$N_f = n_f^3$ denotes the constant total number of grid points in Fourier space. n_s is the octree parameter denoting the maximally allowed number of points inside a leaf node.

- Construction of the octree requires iterating over all $2N$ points at each of the $O(\log(N))$ levels of the tree. This gives $O(N\log(N))$ work. However the observed time for the tree construction is negligible to the actual DMK algorithm, indicating a very small prefactor.
- Calculating the far-field contribution using type 1 and 2 NUFFT can be done in $O(N_f \log(N_f) + N\log^3(1/\epsilon))$ work [2].
- The outgoing expansions are calulated at each of the $O(\log(N))$ levels of the tree. Each level takes $O(N + N_f \log(N_f))$ work [2], making this a $O(N\log(N))$ computation with a precision dependent constant.
- The incoming expansion is calculated for each of the $O(N)$ internal nodes of the tree, each taking $O(N_f)$ work.
- The back-transform of the difference correction terms has to be done at each of the $O(\log(N))$ levels of the tree. Each level takes $O(N + N_f \log(N_f))$ work [2], making this a $O(N\log(N))$ computation with a precision dependent constant.
- The residual term for each point considers at most $O(27n_s)$ other points (a node can have at most 27 colleauges). This gives $O(N)$ work.

This gives a total asymptotic runtime of $O(N\log(N))$.

Chapter 5

Implementation

The octree and the algorithm described above were both implemented into the Independent Parallel Particle Layer library [9]. IPPL is a C++ library for particle mesh methods that makes use of Kokkos [13], a performance portability library. The octree implementation was partially based on an existing, open-source C++ octree library [10], however, it had to be completely rewritten and amended to make it compatible with IPPL. This chapter presents the key parts of the implementation of the algorithm.

5.1 IPPL

The Independent Parallel Particle Layer is a C++ library that focuses on Particle-Mesh methods. It uses the Kokkos C++ library to overcome the problem of performance portability. This requires the use of specific Kokkos types and methods.

```
Kokkos::parallel_for("Calculate u_hat = g_hat * w", field_g.getFieldRangePolicy(),
    KOKKOS_LAMBDA(const int i, const int j, const int k){

        // Convert index (i,j,k) to frequency (k_x,k_y,k_z)
        const double kx = -static_cast<double>(nf/2) + i;
        const double ky = -static_cast<double>(nf/2) + j;
        const double kz = -static_cast<double>(nf/2) + k;

        // Calculation of w(k)
        double kabs = Kokkos::sqrt(kx * kx + ky * ky + kz * kz);
        double w =  Kokkos::pow(Kokkos::numbers::pi, -2) *
                    Kokkos::pow( (Kokkos::sin(Ct * kabs * 0.5) / kabs) , 2) *
                    Kokkos::exp(-0.25 * Kokkos::pow(kabs * sig0_m,2));

        // Component wise multiplication
        uview(i,j,k) = w * gview(i,j,k);

    });
```

Listing 5.1: Kokkos parallel for-loop that iterates over an IPPL field type. This code calculates the far-field contribution in Fourier space.

Kokkos dispatches the work to available resources within a single MPI [8] rank.

5.2 Octree Implementation

The following section shows the high-level code for the octree construction and balancing described in sections 3.2 and 3.3. Construction uses a helper array that stores all points sorted according to the level $l = L_{max}$ leaf node they belong to.

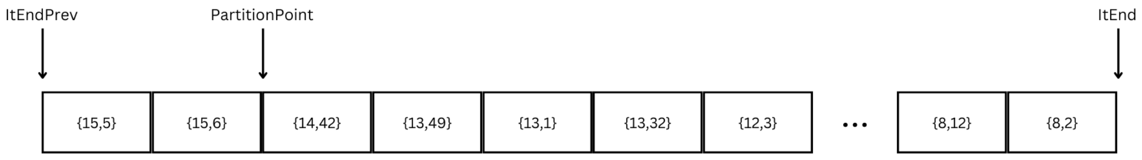

Figure 5.1: Example of a helper array. The first value of each pair is the node's key, and the second is the ID of the point belonging to that node. The pairs are sorted in descending key order, meaning that points belonging to the same node are next to each other in the array. The partition points separate the nodes in the aid array.

```
// addNodes: Function as presented in section 3.2
// nodeParent: Parent node object
// kParent: Parent node key
// itEndPrev: Current location in helper array
// itEnd: End of helper array
// idLocationBegin: Morton Id of first new child node
// nDepthRemain: Remaining allowed depth (L_max - depth)
void addNodes(OrthoTreeNode& nodeParent, morton_node_id_type kParent,
    auto& itEndPrev, auto const& itEnd, morton_node_id_type idLocationBegin,
    depth_type nDepthRemain){

    // Number of points passed to current node
    const auto nElement = static_cast<size_t>(itEnd - itEndPrev);

    // Indicates that leaf has been reached -> Fill points into node
    if(nElement <= this->maxelements_m || nDepthRemain == 0){
        nodeParent.vid_m.resize(nElement);
        std::sort(itEndPrev, itEnd);
        std::transform(itEndPrev, itEnd, nodeParent.vid_m.begin(),
            [](auto const item){return item.first;});
        itEndPrev = itEnd;
        return;
    }

    --nDepthRemain;

    // Bitvalues for Morton encoding
    auto const shift = nDepthRemain * dim_m;
    auto const nLocationStep = morton_node_id_type{1} << shift;
    auto const flagParent = kParent << dim_m;

    // Iteration over aid array to further refine the node
    while(itEndPrev != itEnd){

        // Calculate key of child
        auto const idChildActual =
            morton_node_id_type((itEndPrev->second - idLocationBegin) >> shift);
        // Find the partition point between child nodes
        auto const itEndActual =
            std::partition_point(itEndPrev, itEnd, [&](auto const idPoint)
            {
                return idChildActual ==
                    morton_node_id_type((idPoint.second - idLocationBegin)
                        >> shift);
            });
        auto const mChildActual = morton_node_id_type(idChildActual);

        // Global Morton id of child
        morton_node_id_type const kChild = flagParent | mChildActual;
        morton_node_id_type const idLocationBeginChild =
            idLocationBegin + mChildActual * nLocationStep;

        // Create child
        OrthoTreeNode& nodeChild = this->createChild(nodeParent, kChild);
        nodeChild.parent_m = kParent;

        // Start process for next child
        this->addNodes(nodeChild, kChild, itEndPrev, itEndActual,
            idLocationBeginChild, nDepthRemain);

    } // end while
```

Listing 5.2: addNodes function to recusively build the tree starting from the root node

As described in section 3.3, the tree balancing algorithm fills all the leaf nodes into a queue. A leaf is only popped from the queue once it is balanced, and any new leaf nodes created during this process are added to the queue.

```
// BalanceTree: Function as presented in section 3.3
// positions: Vector of all point's positions
void BalanceTree(Kokkos::vector<position_type> positions){

    // Queue for leaves
    std::queue<morton_node_id_type> unprocessedNodes = {};
    Kokkos::vector<morton_node_id_type> leafNodes = this->GetLeafNodes();

    // Sort leaf nodes in descending key order -> deepest leaves processed first
    std::sort(leafNodes.begin(), leafNodes.end(),
        [](morton_node_id_type l, morton_node_id_type r){return l>r;});

    // Push to queue
    for(unsigned int idx=0; idx<leafNodes.size(); ++idx) {
        unprocessedNodes.push(leafNodes[idx]);
    }

    // Continue until queue is empty <==> All leaves are balanced
    while(!unprocessedNodes.empty()){

        // Certain leaf nodes in the queue might have been
        // refined by previous leaves in the queue
        if (GetNode(unprocessedNodes.front()).IsAnyChildExist()) {
            unprocessedNodes.pop();
        }

        bool processed = true;

        // Get keys of all possible neighbors
        Kokkos::vector<morton_node_id_type> potentialNeighbours =
            this->GetPotentialColleagues(unprocessedNodes.front());



        // Check all potential neighbors
        for(unsigned int idx=0; idx<potentialNeighbours.size();++idx){

            // If the neighbor has the same parent it's a sibling
            if ((potentialNeighbours[idx]>>dim_m)
                == (unprocessedNodes.front()>>dim_m)) continue;

            // If the parent of the neighbor exists,
            // the difference in depth is at most 1
            else if (nodes_m.exists(potentialNeighbours[idx] >> dim_m)) continue;

            // If the parent of the potential neighbor does not exist,
            // The difference in depth is more than 1
            else {

                // The current node is not balanced and stays in the queue
                processed = false;

                // Find the next existing ancestor of the potential neighbor
                morton_node_id_type ancestor =
                    this->GetNextAncestor(potentialNeighbours[idx]);

                // Refine that ancestor
                Kokkos::vector<morton_node_id_type> NewNodes =
                    this->RefineNode(nodes_m.value_at(nodes_m.find(ancestor)),
                        ancestor, positions);

                // Add newly created leaves to the queue
                for (unsigned int idx=0; idx<NewNodes.size(); ++idx) {
                    unprocessedNodes.push(NewNodes[idx]);
                }

            }
        }

        // Leaf is balanced and can be popped from the queue
        if(processed) unprocessedNodes.pop();

    }

}
```

Listing 5.3: Function for balancing the octree

Note that using `std::sort`, `std::queue`, and `std::partition\_point` means that this code is not fully portable. When writing this code, the portable Kokkos equivalents were still experimental and presented issues. In the future, changing the namespace from `std::` to `Kokkos::` should resolve this.

5.3 Non-Uniform Fast Fourier Transform

To compute the far-field and level-wise difference contribution defined in chapter 4, Fourier transforms between discrete points in real space and the points in Fourier space are needed. The regular Fast Fourier Transforms cannot be used since the points are not uniformly spaced. Non-uniform Fast Fourier Transforms, short NUFFTs, are required. For this work, the FINUFFT C++ library [3][1] is used. It implements the **Type 1** NUFFT as the Fourier transform of a set of non-uniformly distributed, discrete points onto a uniform integer grid in Fourier space

$$\hat{f}(\boldsymbol{k}) = \sum_j e^{-i\boldsymbol{k}\cdot\boldsymbol{x}_j}\rho_j \quad \text{for} \quad \boldsymbol{k} \in [-K, -K+1, ..., -1, 0, 1, ..., K-1, K]^3, \quad \boldsymbol{x}_j \in \mathbb{R}^3\,, K \in \mathbb{N}\,. \tag{5.1}$$

Its adjoint, the **type 2** NUFFT, maps a uniform grid in Fourier space onto arbitrary points in real space.

$$c(\boldsymbol{x}) = \sum_j e^{i\boldsymbol{k}\cdot\boldsymbol{x}_j}\hat{f}(\boldsymbol{k}) \quad \text{for} \quad \boldsymbol{k} \in [-K, -K+1, ..., -1, 0, 1, ..., K-1, K]^3, \quad \boldsymbol{x}_j \in \mathbb{R}^3\,, K \in \mathbb{N}\,. \tag{5.2}$$

This library constrains the grid points to only take integer values and the discrete points in real space to be within the interval $[-3\pi, 3\pi]$ in each dimension. For this work, an IPPL wrapper for the FINUFFT transforms that can transform between IPPL field types and IPPL particle types is used.

```cpp
// Initialize the index space of the field type
int nf = static_cast<int>(Kokkos::ceil(4 * Kokkos::log(1/eps_m)));
ippl::Vector<int, dim> pt = {nf, nf, nf};
ippl::Index I(pt[0]);
ippl::Index J(pt[1]);
ippl::Index K(pt[2]);
ippl::NDIndex<dim> owned(I, J, K);

// Specify IPPL field layout
std::array<bool, dim> isParallel;
isParallel.fill(false);
ippl::FieldLayout<dim> layout(MPI_COMM_WORLD, owned, isParallel);


// Initialize the grid origin and spacing
// !SPACING OF 1 IS REQUIRED BY FINUFFT
vector_type hx = {1.0, 1.0, 1.0};
vector_type origin = {
    -static_cast<double>(nf/2),
    -static_cast<double>(nf/2),
    -static_cast<double>(nf/2)
};
mesh_type mesh(owned, hx, origin);

// Initialize field object
fourier_field_type field(mesh, layout,0);
field = 0;

// Initialize NUFFT object
ippl::ParameterList fftParams;
fftParams.add("use_finufft_defaults", true);
int type = 1;
std::unique_ptr<nufft_type> nufft1 = std::make_unique<nufft_type>(layout,
    sources_m.getTotalNum(), type, fftParams);

// Perform NUFFT of the sources onto the field
nufft1->transform(sources_m.R, sources_m.rho, field);
```

Listing 5.4: Type 1 NUFFT of IPPL points onto an IPPL field

5.4 DMK Implementation

The restriction to integer grid points in Fourier space (by the FINUFFT library) poses a difficulty in the computation of the level-wise difference correction. Recall section 4.4, we want to calculate

$$u_{\text{diff},l}(\boldsymbol{x}_i) \approx \sum_{\boldsymbol{k}} e^{i\boldsymbol{k}\cdot(\boldsymbol{x}_i - \boldsymbol{c}(\boldsymbol{B}))} \sum_{S\in\mathcal{C}(B)} e^{i\boldsymbol{k}\cdot(\boldsymbol{c}(\boldsymbol{B})-\boldsymbol{c}(\boldsymbol{S}))} \frac{1}{(2\pi)^3}\hat{D}_l(\boldsymbol{k}) \sum_{\boldsymbol{y}_j\in S} e^{i\boldsymbol{k}\cdot(\boldsymbol{c}(\boldsymbol{S})-\boldsymbol{y}_j)}\rho_j\,, \tag{5.3}$$

where $\boldsymbol{k} \in [-h_l n_f, ..., h_l n_f]^3$ are a constant number of points n_f^3 across all levels of the octree. As illustrated in figure 4.3, the Fourier-space difference kernel's support increases with the tree's depth, requiring larger integration boundaries. To maintain a constant number of quadrature points (grid points), the spacing between points h_l needs to increase with the depth. However, as explained in section 5.3, the FINUFFT library prescribes an integer grid with unit spacing. A workaround to this can be achieved by first re-writing equation 5.3 as

$$u_{\text{diff},l}(\boldsymbol{x}_i) \approx \sum_{\boldsymbol{m}} e^{ih_l\boldsymbol{m}\cdot(\boldsymbol{x}_i - \boldsymbol{c}(\boldsymbol{B}))} \sum_{S\in\mathcal{C}(B)} e^{ih_l\boldsymbol{m}\cdot(\boldsymbol{c}(\boldsymbol{B})-\boldsymbol{c}(\boldsymbol{S}))} \frac{1}{(2\pi)^3}\hat{D}_l(h_l\boldsymbol{m}) \sum_{\boldsymbol{y}_j\in S} e^{ih_l\boldsymbol{m}\cdot(\boldsymbol{c}(\boldsymbol{S})-\boldsymbol{y}_j)}\rho_j\,, \tag{5.4}$$

where $\boldsymbol{m} \in [-n_f, ..., n_f]^3$. Letting the scaling factor h_l act on the positions of the Fourier transform instead of the integer grid $\boldsymbol{m}$, this can be understood as performing a Fourier transform of positions, scaled by h_l, onto an integer grid. The dilation property of Fourier Transforms in three dimensions (equation 2.5) gives

$$h_l^3\mathcal{F}\{\rho(h_l r)\}(\boldsymbol{m}) = \mathcal{F}\{\rho(r)\}(h_l\boldsymbol{m})\,. \tag{5.5}$$

This means evaluating the Fourier transform of sources with positions scaled by the factor h_l at integer locations $\boldsymbol{m} \in [-n_f, ..., n_f]^3$ and multiplying by h_l^3 is equivalent to performing the normal Fourier transform and evaluating at $\boldsymbol{k} \in [-h_l n_f, ..., h_l n_f]^3$.

```cpp
// Calculation of the outgoing expansion
// Get node's center
ippl::Vector<double,dim> center = tree_m.GetNode(key).GetCenter();

// Create particles shifted relative to the nodes center and scaled by hl
particle_type relSources(PLayout);
relSources.create(idSources.size());
Kokkos::parallel_for("Create relative particles", idSources.size(),
KOKKOS_LAMBDA(unsigned int i){
    relSources.R(i) = hl * (sources_m.R(idSources[i]) - center);
    relSources.rho(i) = sources_m.rho(idSources[i]);
});

// Initialize NUFFT1
int type = 1;
std::unique_ptr<nufft_type> nufft1 = std::make_unique<nufft_type>(layout,
    relSources.getTotalNum(), type, fftParams);

// Initialize field for Fourier transform
fourier_field_type fieldPhi(mesh, layout, 0);
fieldPhi = 0;

// Perform type 1 NUFFT with scaled positions
nufft1->transform(relSources.R, relSources.rho, fieldPhi);
```

Listing 5.5: Calculation of the outgoing expansion via the dilation trick.

The incoming expansion is calculated using Kokko's parallel loop to perform the element-wise multiplication of the Fourier-transformed sources with the Fourier-transformed difference kernel.

```
 1 // Calculate incoming expansion
 2 // Kokkos loop over integer frequencies m = [i,j,k]
 3 Kokkos::parallel_for("Calculate incoming expansion", fieldPsi.getFieldRangePolicy()
     ,
 4 KOKKOS_LAMBDA(const int i, const int j, const int k){
 5 
 6     // Get scaled frequencies k = hl * m
 7     const double kx = (-static_cast<double>(nf/2) + i) * hl;
 8     const double ky = (-static_cast<double>(nf/2) + j) * hl;
 9     const double kz = (-static_cast<double>(nf/2) + k) * hl;
10 
11     // Calculate difference kernel
12     double w =  Kokkos::pow(Kokkos::numbers::pi*2,-3) *  D(depth, kx, ky, kz);
13 
14     // Dot product of (kx,ky,kz) and (center-difference)
15     double t = kx * delta[0] + ky * delta[1] + kz * delta[2];
16 
17     // i
18     Kokkos::complex<double> imag;
19     imag.real() = 0; imag.imag() = 1;
20 
21     PsiView(i,j,k) += w * Kokkos::exp(imag * t) * PhiView(i,j,k)
22         * Kokkos::pow(hl,3);
23 
24 
25 });
```

Listing 5.6: Kokkos loop for calculating the incoming expansion. The outgoing expansion is evaluated at integer frequencies in accordance with FINUFFT. Usage of the dilation property requires multiplying with an additional factor h_l^3 in line 22.

Chapter 6

Numerical Results

This chapter discusses the numerical results of the implemented DMK algorithm. The tests were performed on a local machine with an Intel Core i9-10900K @3.70GHz CPU. For the test problem, points are uniformly distributed across the domain $\Omega = [0,1]^3$, and the sources follow a Gaussian with mean $\boldsymbol{\mu} = (0.5, 0.5, 0.5)$ and standard deviation $\sigma = 0.05$

$$\rho(\boldsymbol{r}) = \frac{1}{\sqrt{8\pi^3}\sigma^3} e^{-(\boldsymbol{r}-\boldsymbol{\mu}^2)/(2\sigma^2)} . \tag{6.1}$$

The solver is compared to the explicit computation of the convolution with the Green's function, as described in equation 4.2. In the first part, the error of the DMK solver is analyzed for different problem sizes and choices of solver parameters n_s and precision ϵ. The performance is benchmarked both serially and with ten threads on a single MPI rank and the relative speedup is analyzed.

6.1 Convergence

This section compares the relative error of the solver's solution to the explicitly calculated solution. The error metric used is the Mean Relative Error defined as

$$\text{error} = \frac{1}{N} \sum_{i=1}^{N} \frac{|\rho_i - \rho(\boldsymbol{x}_i)|}{|\rho(\boldsymbol{x}_i)|}, \tag{6.2}$$

where ρ_i is the solver's solution for target point $\boldsymbol{x}_i$ and $\rho(\boldsymbol{x}_i)$ is the true value. Figure 6.1 shows the error of the DMK solver for chosen parameters $n_s = 700, 1000, 2000, 4000$ and $\epsilon = 10^{-3}, 10^{-6}$ over increasing problem sizes.

6.2 Solver Timings

The runtime of the DMK solver is tested for the parameters $n_s = 700, 1000$ with $\epsilon = 10^{-3}$. It is done serially and, in parallel, using 10 cores on a single MPI rank. The results are shown in figures 6.2 6.3 6.4 6.5.

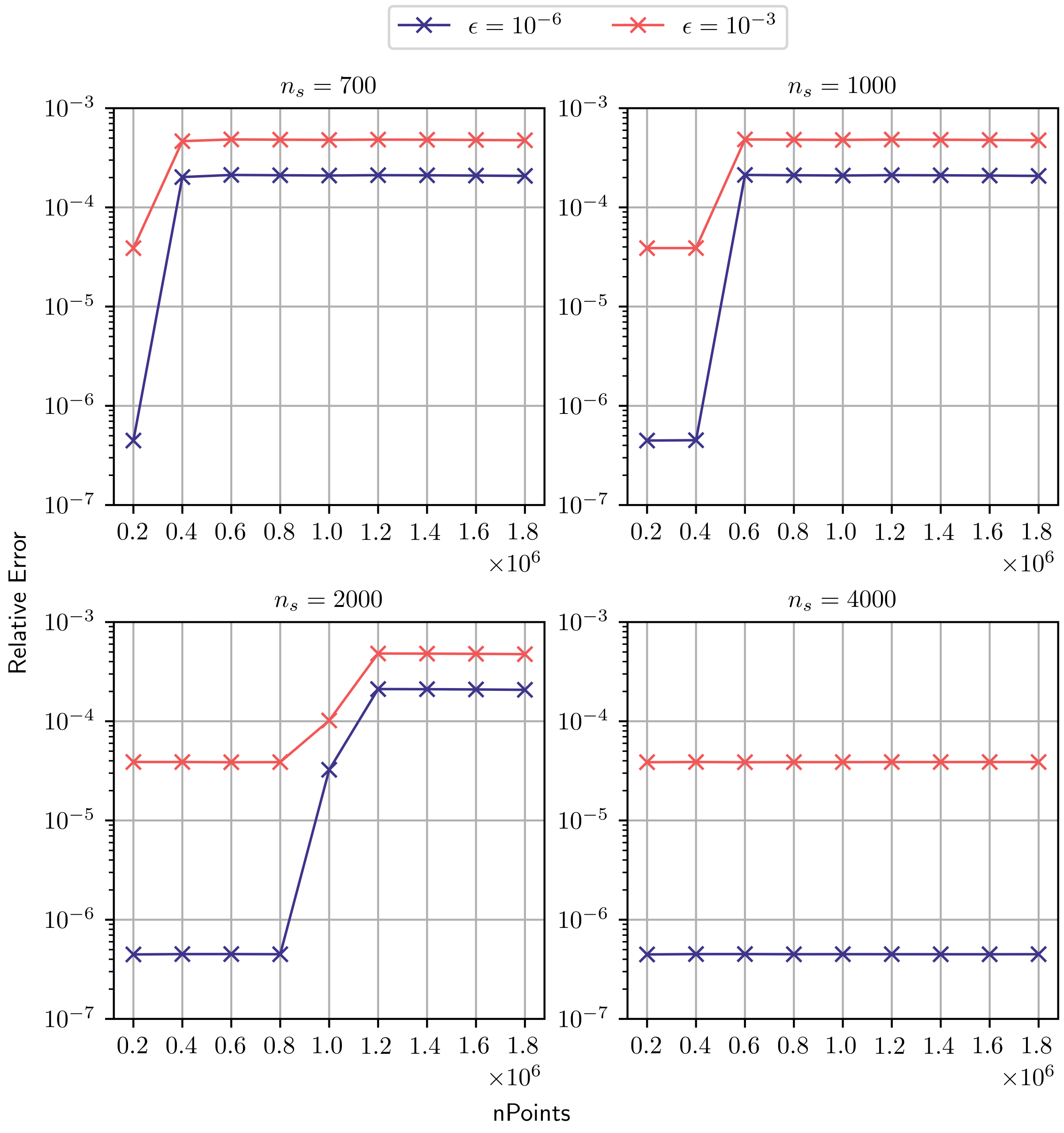

Figure 6.1: The relative error (equation 6.2) of the solver for $n_s = 700, 1000, 2000, 4000$ and $\epsilon = 10^{-3}, 10^{-6}$ over the problem sizes $0.2 - 1.8 \cdot 10^6$. For $n_s = 700, 1000, 2000$ a sharp jump in the error occurs at problem sizes $N = 0.4, 0.6, 0.8 \cdot 10^6$, respectively. The parameter n_s controls the number of points per leaf node, meaning a larger n_s produces a more shallow tree. In all three cases, the jump occurs once the number of points N is large enough for the tree to reach depth $l = 4$. For $n_s = 4000$, the tree never exceeds depth $l = 3$, indicating that this jump is related to the depth of the tree. Apart from the jump, the error stays constant for different problem sizes. It is unclear what the reason for this behavior is; however, a possible explanation for this is a round-off error due to large differences in the magnitude of summed terms. Despite the jump, the prescribed precision $\epsilon = 10^{-3}$ is achieved in all cases. This is not the case for $\epsilon = 10^{-6}$, where the accuracy is achieved only when the tree depth does not exceed $l = 3$.

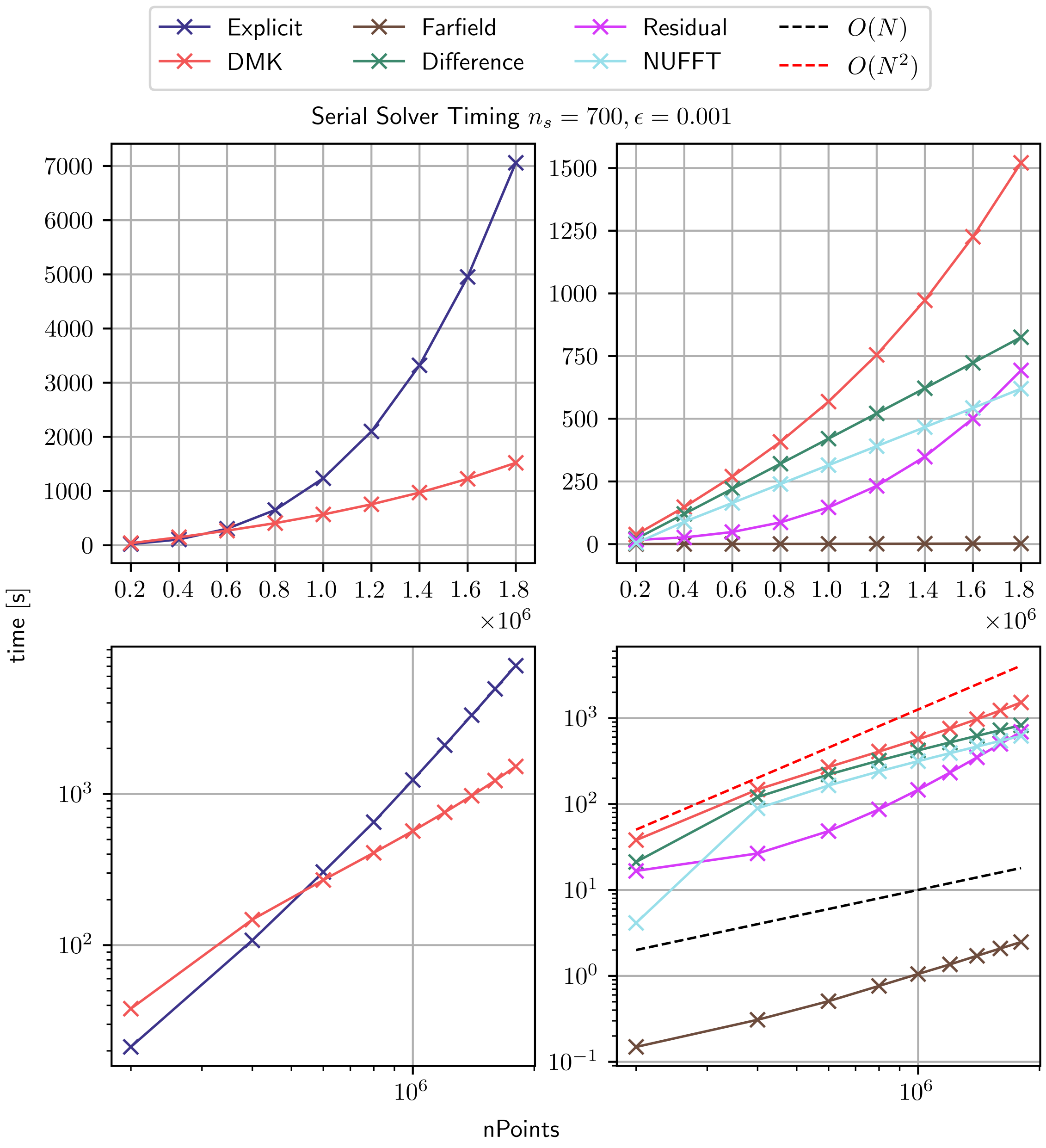

Figure 6.2: The top-left plot compares the total runtime of the solver to that of the explicit solution. At $N = 0.6 \cdot 10^6$ the solver gains a increasing advantage. The top-right plot shows the breakdown of different parts of the solver. The computation of the level-wise difference corrections (green) and the residual contribution (purple) are roughly balanced, while the far-field contribution (brown) takes comparatively no time at all. The light blue line shows that a significant part of the difference computation is taken up by calculating NUFFTs. The bottom two plots show the same data presented in log-log plots, where, apart from the residual contribution, all parts fall between the asymptotic lines $O(N)$ and $O(N^2)$. The reason for the deviation from the theoretical runtime of the residual contribution is the choice of n_s, which was chosen to balance the cost of residual and difference computation. The parameter is large relative to the total number of points at these problem sizes, meaning the calculated linear runtime $O(n_s N)$ behaves more like $O(N^2)$. When n_s is chosen small relative to the problem size, the theoretical linear runtime would be observed.

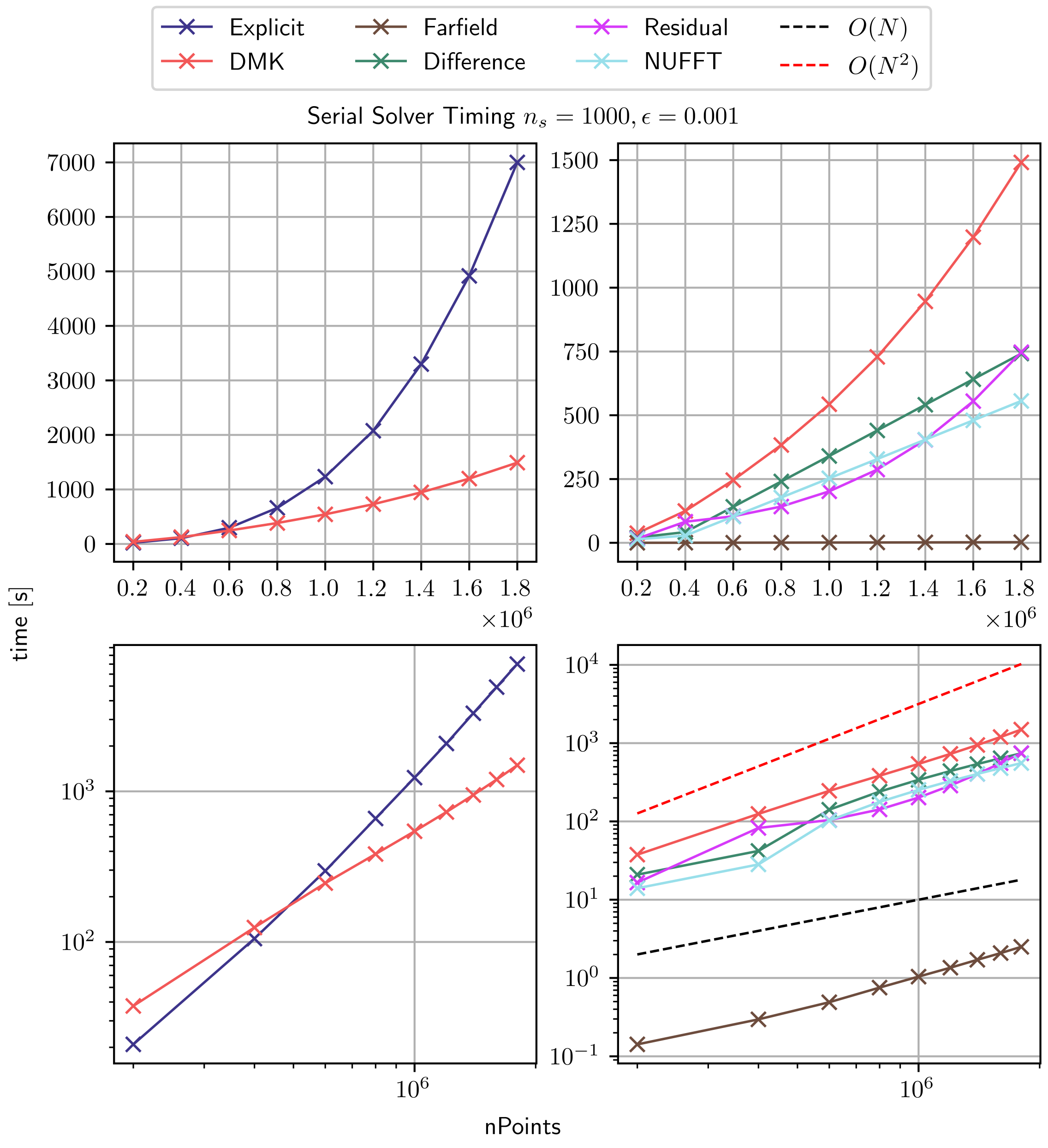

Figure 6.3: For $n_s = 1000$ the behavior is qualitatively the same as for figure 6.2. The cost of the residual contribution relative to the rest is larger. Since this choice of n_s produces a tree with more points per leaf node, this change is expected.

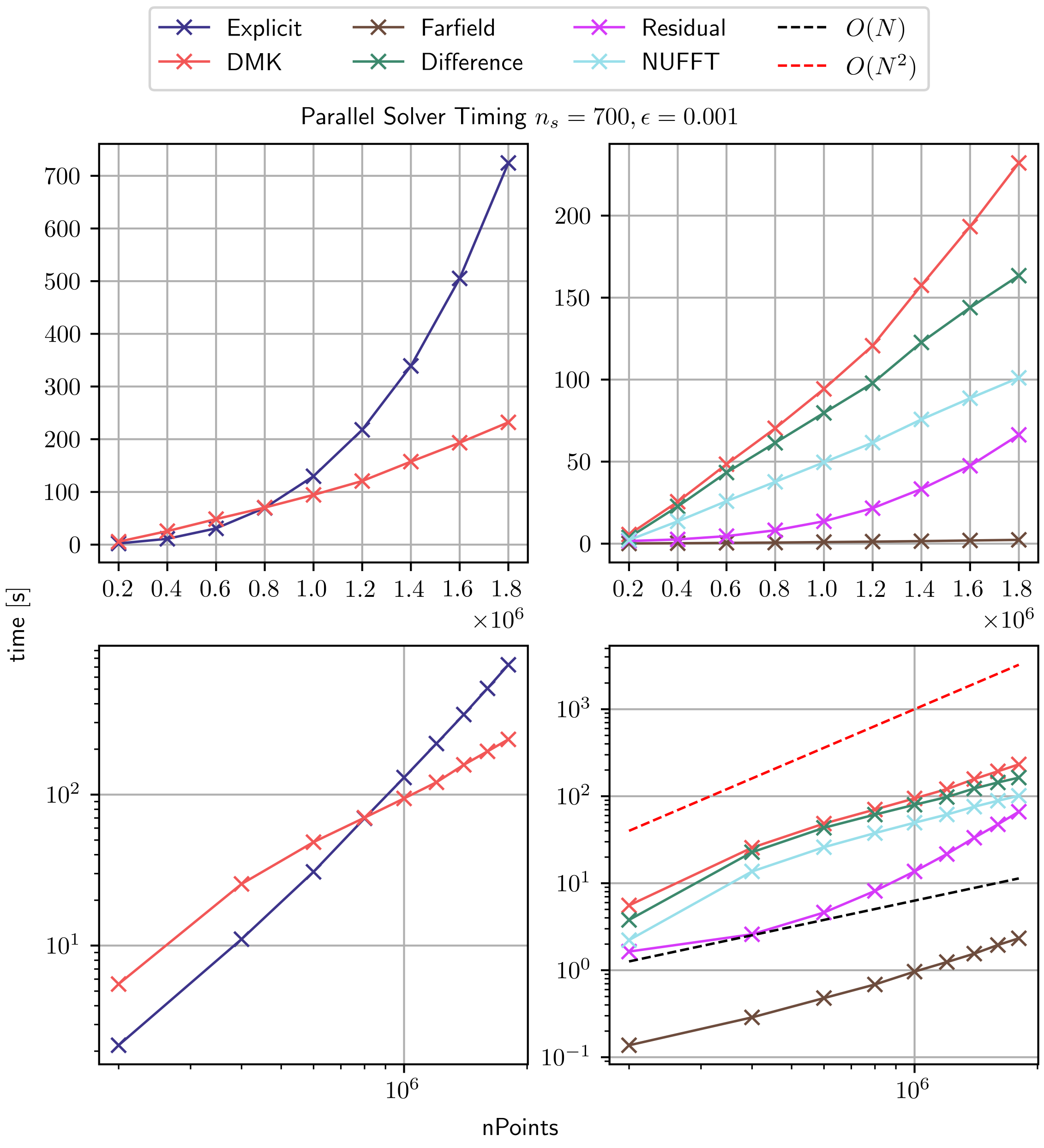

Figure 6.4: The layout is the same as for the serial timing plots. It can be observed that it takes larger problems for the solver to become faster than in the serial case. This is because of the difference in the scaling efficiency of the explicit solution and the DMK solver. The explicit solution is implemented via independent for-loop and, as a result, has perfect scaling. The total work is the same as for the serial case since the same parameters n_s and ϵ are chosen. However, the top-right plot shows that the relative time taken up by the difference and residual kernels differ significantly. This is a result of differing scaling efficiency between the different contributions; see figures 6.6 and 6.7

.

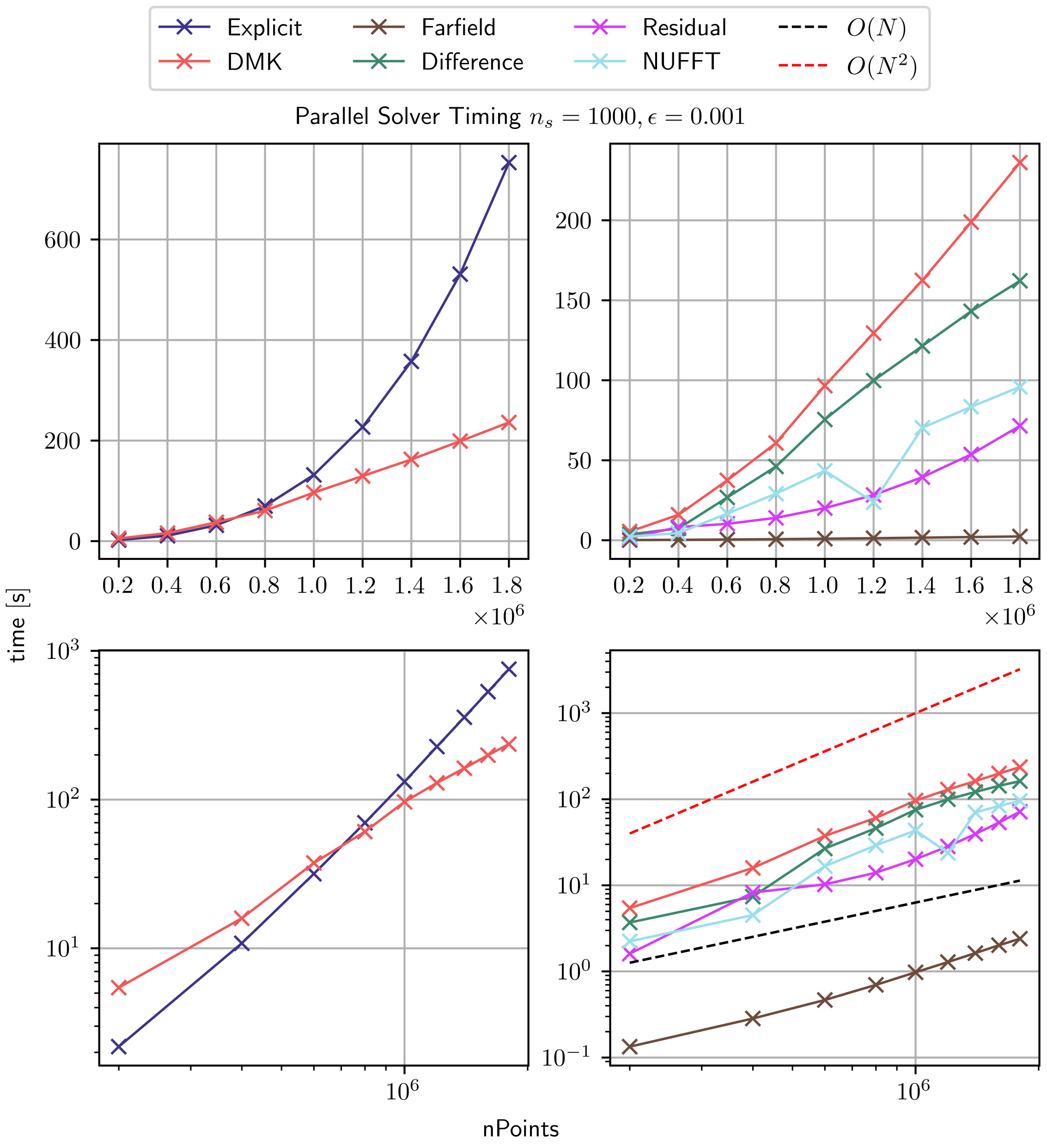

Figure 6.5: We can observe the same qualitative behavior for this set of parameters, as in figure 6.4. Like the serial case, choosing a large n_s means that the residual kernel cost behaves closer to $O(N^2)$ for this problem size.

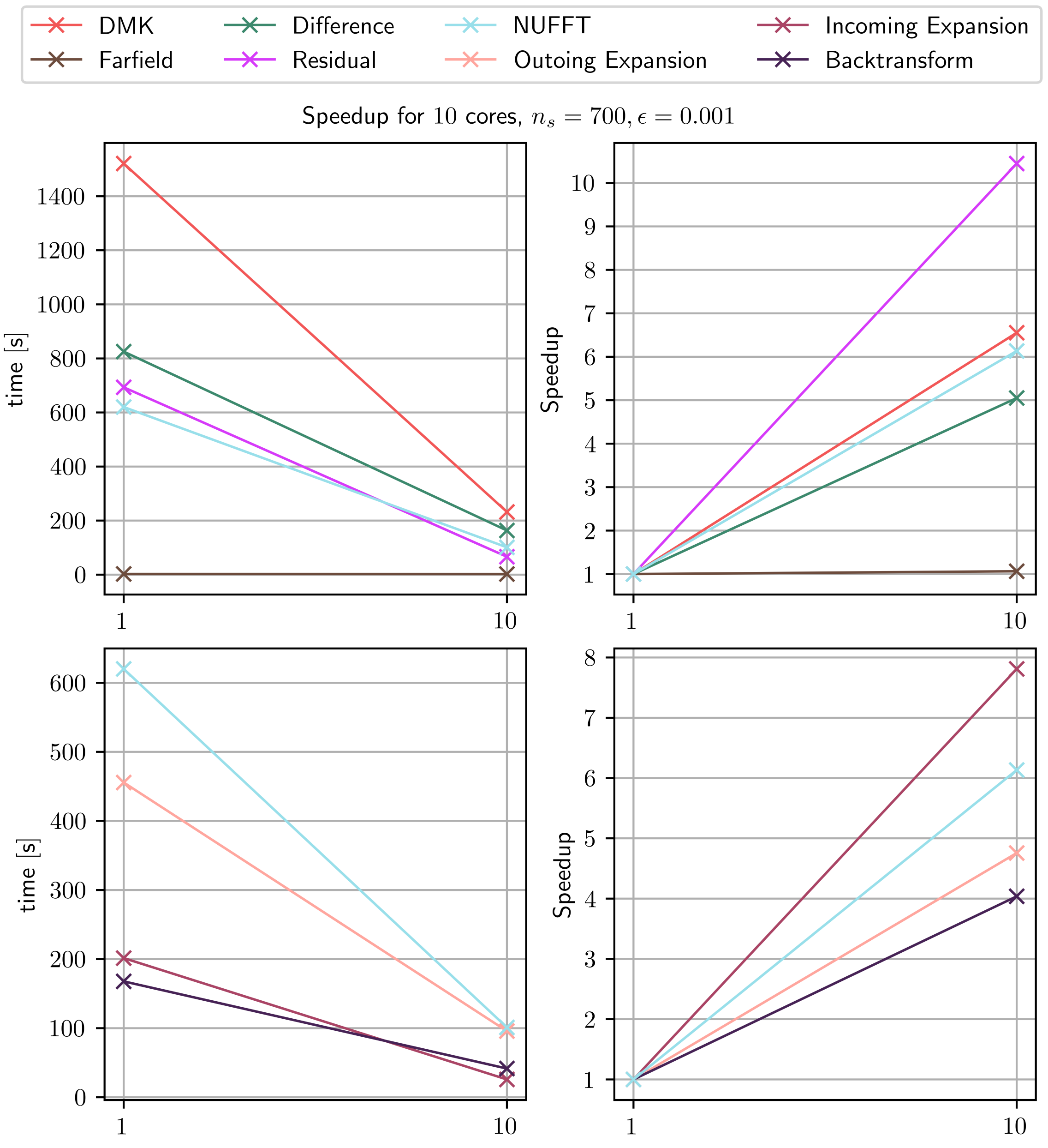

Figure 6.6: Comparison of runtimes and relative speedups for 1 and 10 cores on a single MPI rank. We can observe that the difference kernel contribution scales with 50% efficiency. As demonstrated in figures 6.2 - 6.5, NUFFTs make up the majority of the its cost. The residual kernel does not perform NUFFTs and requires only independent for-loops, giving perfect scaling. The far-field kernel's total cost is so small that the cost incurred by launching multiple threads outweighs the reduced computation time. Overall, the DMK algorithm shows roughly 65% efficiency. The bottom-right plot shows a breakdown of the parts of the difference kernel. The two parts that require NUFFTs (outgoing expansion and back-transform) both have poor scaling, while the point-wise multiplication of fields (incoming expansion) enjoys almost 80% scaling.

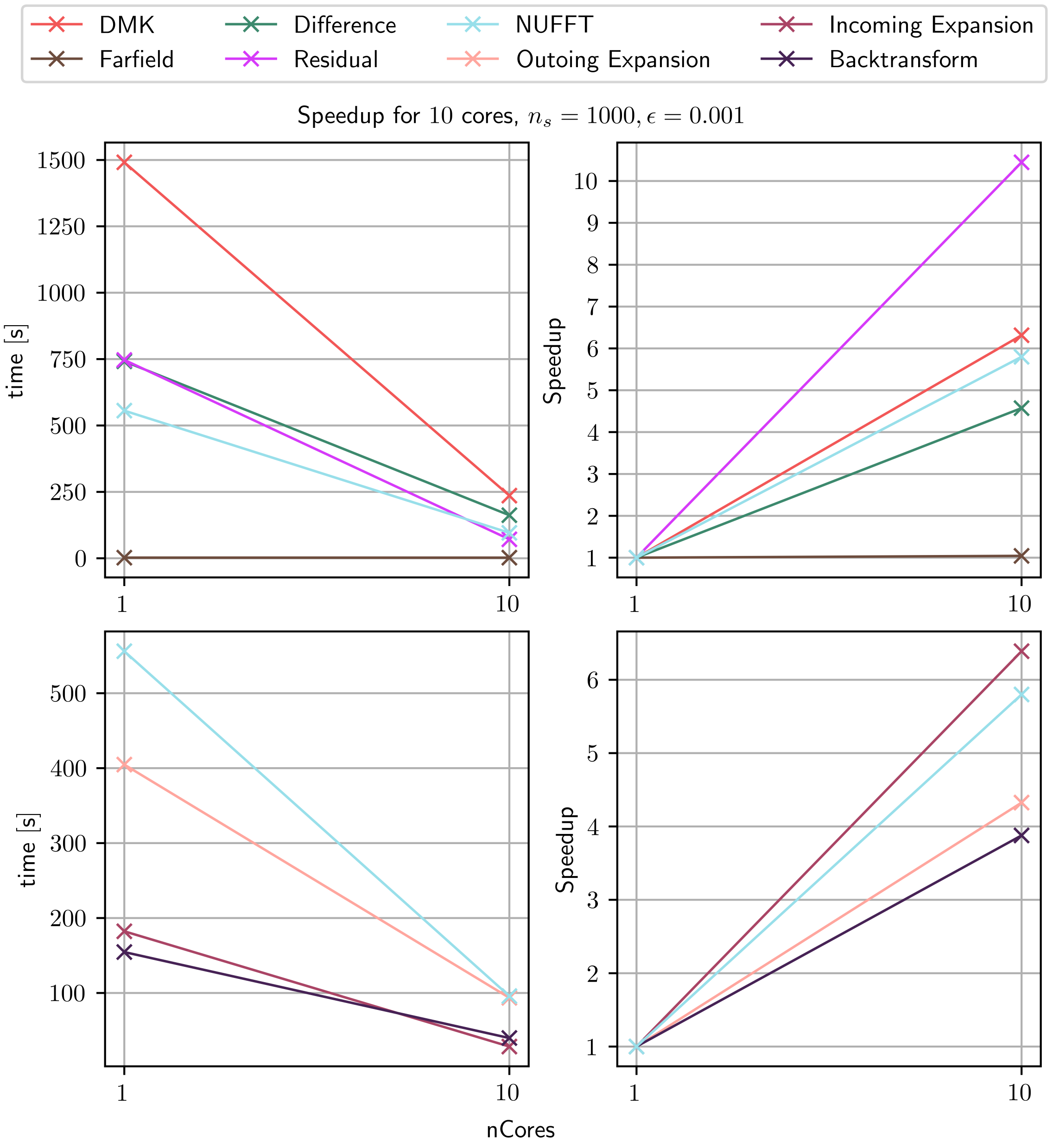

Figure 6.7: Compared to figure 6.6 the difference kernels has slightly worse scaling. With a different choice of n_s, the constructed tree changes. Potentially, in a way that makes the size and number of NUFFTs and the point-wise multiplications less amenable to parallelism. The residual kernel still scales perfectly, and the overall solver shows roughly 63% scaling.

Chapter 7

Conclusion and Future Work

In this work a dual-space multilevel kernel-splitting algorithm, first introduced by Jiang et al.(2023) [6], was implemented into the IPPL C++ library [9]. This first required the implementation of an octree, which is used to generate a multilevel and adaptive mesh based on the point distribution inside the computational domain. In serial, the DMK solver gains an advantage at $6 \cdot 10^5$ points for the chosen parameter configuration. This advantage grows with larger problem sizes. When using 10 threads on a single MPI rank, this can be observed after $8 \cdot 10^5$ points.

There are many ways to build upon this first implementation in future work. We saw that the algorithm's far-field calculation takes almost no time relative to the other contributions. The cost of the residual terms can be directly controlled via the parameter n_s. That leaves the computation of the level-wise difference corrections. Most of the time is taken up by the NUFFTs, so the performance depends on the underlying FINUFFT C++ implementation. FINUFFT can use multiple threads to compute a NUFFT but cannot compute multiple NUFFTs simultaneously. However, for this algorithm, the size of the individual transforms is small and only linearly depends on the number of points, as explained in section 4.6. It would be more efficient to have each thread perform its own transform. This can be achieved by using the vectorization capabilities of FINUFFT. It comes with the restriction that the set of non-uniform points must be the same for each transform. The updated difference contribution at each level l would be:

1. For all internal nodes at level l: interpolate the sources onto a set of grid points inside the node.
2. Calculate the outgoing expansion Φ_l for all the nodes with a vectorized NUFFT.
3. Calculate the incoming expansion Ψ_l for all nodes normally.
4. Perform the vectorized NUFFT back onto the grid points in each node.
5. Interpolate the values at the grid points onto the target points.

This approach could potentially make the calculation of the difference kernel more efficient, allowing for a choice of n_s that yields overall better performance.

The original DMK paper proposes a two-pass version that builds on the idea of interpolating points onto a grid. The upward pass forms the interpolation of sources onto a Chebychev grid at the leaf level. Using interpolation this set of proxy charges is mapped onto a set of Chebychev grid points at all higher levels. The downwards pass is very similar to the originally proposed DMK algorithm. However, now the N points do not have to be accessed at all $\log(N)$ levels of the tree, allowing for a total asymptotic runtime of $O(N)$.

With IPPL's purpose as a framework for performing electrostatic simulations in mind, many optimizations can be made regarding the octree. For simulations where particles move around the domain, the octree needs to be updated continuously, precisely when a large number of particles move between leaves. The octree-based domain decomposition naturally extends to large simulations, where each MPI rank gets its own branch of the tree. This requires local-to-global parallel balancing methods such as those described in Sundar et at. [12].

Acknowledgements

First and foremost, I would like to express my deepest thanks to my supervisor, Dr. Andreas Adelmann, for allowing me to do my Master's thesis with the AMAS group at PSI. Thank you for supporting and guiding me throughout the project.

I would also like to extend my gratitude to Sonali Mayani for providing valuable help and guidance with all parts of the project. Thank you, I really appreciate your help.

I would like to acknowledge Dr. Sriramkrishnan Muralikrishnan for supporting me with various implementation issues. Your patience and advice are deeply appreciated.

I would also like to thank the other members at the weekly meetings for their support and ideas.

Bibliography

[1] A. H. Barnett. “Aliasing error of the $\exp(\beta\sqrt{1-z^2})$ kernel in the nonuniform fast Fourier transform”. In: (2020). arXiv: 2001.09405.

[2] Alex H. Barnett, Jeremy F. Magland, and Ludvig af Klinteberg. *A parallel non-uniform fast Fourier transform library based on an ”exponential of semicircle” kernel.* 2019. arXiv: 1808.06736 [math.NA].

[3] Alex H. Barnett, Jeremy F. Magland, and Ludvig af Klinteberg. “A parallel non-uniform fast Fourier transform library based on an ”exponential of semicircle” kernel”. In: (2019). arXiv: 1808.06736.

[4] H. Cheng, L. Greengard, and V. Rokhlin. “A Fast Adaptive Multipole Algorithm in Three Dimensions”. In: *Journal of Computational Physics* 155.2 (1999), pp. 468–498. ISSN: 0021-9991. DOI: https://doi.org/10.1006/jcph.1999.6355. URL: https://www.sciencedirect.com/science/article/pii/S0021999199963556.

[5] L Greengard and V Rokhlin. “A fast algorithm for particle simulations”. In: *Journal of Computational Physics* 73.2 (1987), pp. 325–348. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(87)90140-9. URL: https://www.sciencedirect.com/science/article/pii/0021999187901409.

[6] Shidong Jiang and Leslie Greengard. *A Dual-space Multilevel Kernel-splitting Framework for Discrete and Continuous Convolution.* 2023. arXiv: 2308.00292 [math.NA].

[7] Ludvig af Klinteberg, Davoud Saffar Shamshirgar, and Anna-Karin Tornberg. “Fast Ewald summation for free-space Stokes potentials”. In: *Research in the Mathematical Sciences* 4.1 (Feb. 2017). ISSN: 2197-9847. DOI: 10.1186/s40687-016-0092-7. URL: http://dx.doi.org/10.1186/s40687-016-0092-7.

[8] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0.* June 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[9] Sriramkrishnan Muralikrishnan et al. “Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures”. In: *Proceedings of the 2024 SIAM Conference on Parallel Processing for Scientific Computing (PP).* SIAM. 2024, pp. 26–38.

[10] *Octree/Quadtree/N-dimensional linear tree.* URL: https://github.com/attcs/Octree.

[11] “Schwartz Space”. In: (May 2024). URL: https://en.wikipedia.org/wiki/Schwartz_space.

[12] Hari Sundar, Rahul Sampath, and George Biros. “Bottom-Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel”. In: *SIAM J. Scientific Computing* 30 (Aug. 2008), pp. 2675–2708. DOI: 10.1137/070681727.

[13] Christian R. Trott et al. “Kokkos 3: Programming Model Extensions for the Exascale Era”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[14] "Z-order curve". In: (June 2024). URL: `https://en.wikipedia.org/wiki/Z-order_curve`.

Appendix

Source Files

The work is implemented in a repository forked from the original IPPL [9] repository. It can be found at `https://github.com/rammann/ippl-octree/tree/octreev2`. The source files are located in the `/src/OrthoTree/` directory.

`Orthotree.h` implements the octree structure with all the necessary functionality for the DMK algorithm.
`OrthoTreeParticle.h` implements the discrete IPPL particles; they have a three-dimensional position vector and a value.
`TreeOpenPoissonSolver.h` implements the DMK algorithm.

Compiling

Apart from the dependencies required by IPPL, compiling this project involves first installing the FINUFFT C++ library. The installation is documented at `https://finufft.readthedocs.io/en/latest/install.html`. Once it is installed, the CMake module at `/CMakeModules/FindFINUFFT.cmake` needs to be able to find the FINUFFT installation. To ensure this perform:

```
    export FINUFFT_INCLUDE_DIR [path to finufft.h]
    export FINUFFT_LIBRARY [path to finufft.so]
```

Then, the documented installation process of the IPPL C++ library can be followed.

Tests

The test files are located at `/test/orthotree/`. `TestGaussianTreeSolver.cpp` implements the runtime test of the solver and compares the error to the explicit solution. Once compiled, the executable takes the following arguments:

```
    ./TestGaussianTreeSolver nPoints nPointsPerLeaf --info 5
```

`nPoints` is the number of source points, which equals the number of target points; it is effectively $N/2$. `nPointsPerLeaf` is the maximal number of points per leaf node n_s.

`TestTreeConvergence.cpp` implements the convergence test. Once compiled, the executable takes the following arguments:

```
    ./TestTreeConvergence nPoints nPointsPerLeaf --info 5
```

Convergence is tested for different problem sizes: $N = 2\text{nPoints}, 2 \cdot \text{nPoints}, ..., 9 \cdot \text{nPoints}$.