Contents

1 Foreword

This document summarizes the work completed during Aurélien Gauthier's internship with the CONCACE team, under the supervision of Antoine Gicquel, PhD student in CONCACE. The objective was to explore the Kokkos ecosystem by attempting to implement existing kernels from the ScalFMM library. This document serves as a reflection on that experience and includes code snippets from the developed implementations to highlight some of the features offered by Kokkos. However, note that it is not intended to be a formal guide or comprehensive summary of the official documentation (which can be found here), though pointers to relevant sections are provided. We only explored specific features of Kokkos that suited our needs for a specific target application (detailed in the introduction), and thus, this document does not aim to be an exhaustive overview of the library. Nonetheless, we hope that our experience might be valuable to others interested in learning more about Kokkos. At the time of writing, the latest release of Kokkos is version 4.3.01, please note that some features presented here may change as the API evolves. Additionally, certain results are preliminary and may require further investigation (see the result section). Finally, feedback is warmly welcome!

2 Introduction

Before diving into the heart of the matter, let's briefly introduce the context of the internship.

2.1 Context

We are interested in physical applications where we evaluate pairwise interactions between N bodies (by "bodies" we mean particles, planets, celestial objects, etc.). Each of the N bodies is located at a position $x_i \in \mathbb{R}^n$ ($n = 2$ or $n = 3$ most of the time). Generally, this can be expressed as follows:

$$\Phi_i = \sum_{j=1, i \neq j}^{N} k(x_i, x_j) m_j \quad \forall i \in \{1, \ldots, N\}. \tag{1}$$

In equation (1):

- x_i is the **position** of body i.
- m_i represents the **input** for body i (for example, it could be an electric charge or mass).
- Φ_i represents the **output** for body i (for example, it could be a potential or a force).
- $k(x, y)$ if the **interaction function** (or interaction kernel). This function often has a singularity when x and y are close to each other, but it becomes smooth and decays rapidly as the distance from the singularity increases.

We will now provide some examples of this physical problem.

2.1.1 Electrostatic potential

In this case, we consider a set of N particles, each with a **charge** q_i, and we want to compute the **potential** p_i for each one of them using the following expression:

$$p_i = \frac{1}{4\pi\epsilon_0} \sum_{j=1, i \neq j}^{N} \frac{q_j}{\|x_i - x_j\|} \quad \forall i \in \{1, \ldots, N\}, \tag{2}$$

where ϵ_0 is the vaccum permeability. Thus, the **interaction function** is defined as:

$$k(x, y) = \frac{1}{\|x - y\|}.$$

2.1.2 Gravitational interactions

In this case, we have a set of N celestial bodies (planets, stars, etc. . .) each with a **mass** m_i, and we aim to calculate the **total force** $\vec{F}_i$ acting on each one of them using the following formula:

$$\vec{F}_i = Gm_i \sum_{j=1, i\neq j}^{N} \frac{(x_j - x_i)}{\|x_i - x_j\|^3} m_j \quad \forall i \in \{1, \ldots, N\}, \tag{3}$$

where G is the gravitational constant. Thus, the **interaction function** is defined as:

$$k(x, y) = \frac{x - y}{\|x - y\|^3}.$$

2.2 The Fast Mutipole Method

One can notice that the naive direct evaluation of (1) has quadratic complexity (*i.e.*, (N^2)). The *Fast Multipole Method* (FMM) [**greengard1987fast**] is a hierarchical method used to accelerate the evaluation of (1). It specifically speeds up the computation of **long-range interactions** (*i.e.*, the **distant interactions**) between a large number of bodies. Instead of calculating all pairwise interactions by evaluating (1), which would be obviously prohibitively time-consuming, the FMM groups distant bodies and approximates their interactions. This significantly reduces the computation time. For a more comprehensive, in-depth explanation of the FMM, we direct the reader to the first chapter of [**blanchard2017fast**]. However, a thorough understanding of the FMM is not required to follow this document; this reference is provided simply to give the overall context.

In the FMM algorithm, the space in which the N bodies are located is **hierarchically** divided into **clusters** (or **boxes**) that are nested within one another. This hierarchical division is usually represented by a tree, where each node represents a box (this is a d -tree; in 3D it is called an *octree*, and in 2D, a *quadtree*,. . .). We usually consider that interactions between bodies located in the same box or in neighboring boxes are part of the **near field**, while other interactions are part of the **far field**.

In practice, the FMM separates the computation of the **near field** from the **far field**. As briefly mentioned above, the FMM groups distant bodies and uses approximations to compute the **far field**. However, for the **near field** calculation, approximations are not accurate enough. As a consequence, the interactions between each pair of bodies must be computed

explicitly and precisely (like in expression (1)). This is the purpose of the **P2P** operator, which we will describe in the next paragraph. In the end, the FMM algorithm performs a fast evaluation of (1) (by "fast evaluation" we mean $(N \log N)$ and even (N) with a multi-level strategy).

2.3 The P2P kernel

As we mentioned earlier, the **P2P** (*Particle-to-Particle*) kernel is the part of the FMM algorithm responsible for the direct calculation of interactions between nearby bodies.

There are actually two variants of the **P2P kernel**, but the underlying concept is the same for both: iterating through bodies to update the output of others. In this context, we refer to the bodies whose outputs are being updated as *target bodies*, and those contributing to the interactions as *source bodies*. Note that, in most cases, the *target* and *source bodies* are actually the same.

- **First version:** The **P2P inner** kernel (see Algorithm fig. 1) calculates the interactions between bodies located within the same box B. We can notice that we skip the case where $i = j$ (*i.e.*, self-interaction). This situation corresponds to the one illustrated in Figure fig. 2.

```
/* --- ARGUMENTS ---                                                    */
Input: x: positions of the N bodies in box B
Input: m: inputs of the N bodies in box B
Input: k: interaction function
Output: Φ: outputs of the N bodies in box B

/* --- ALGORITHM ---                                                    */
Procedure P2P-inner(x, m, k, Φ):
    // Loop through target bodies
    for i ← 1 to N do
        // Loop through source bodies
        for j ← 1 to N do
            if i ≠ j then
                Φ[i] ← Φ[i] + k(x[i], x[j]) × m[j]
```

Figure 1: P2P inner algorithm

- **Second version:** The **P2P outer** kernel (see Algorithm fig. 3) calculates the interactions between bodies located in two neighboring boxes.

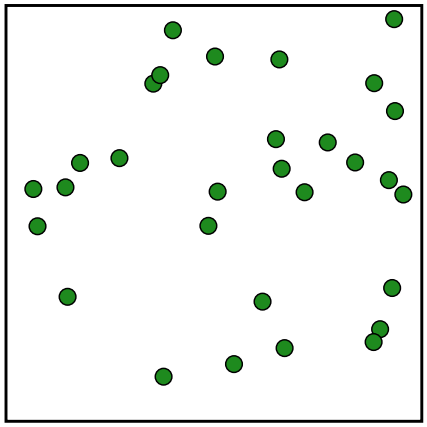

Figure 2: Bodies in a single box (typical situation for the P2P inner kernel)

We consider a first box B_1 with M bodies and a second box B_2 with N other bodies. Each of the bodies in B_1 contributes to update the **output** of the bodies in B_2. This situation corresponds to the one illustrated in Figure fig. 4.

```
/* --- ARGUMENTS ---                                                    */
Input: x: positions of the M bodies in box B_1
Input: y: positions of the N bodies in box B_2
Input: m: inputs
Input: k: interaction function
Output: Φ: outputs of the M bodies in box B_1

/* --- ALGORITHM ---                                                    */
Procedure P2P-outer(x, y, m, k, Φ):
    // Loop through target bodies
    for i ← 1 to M do
        // Loop through source bodies
        for j ← 1 to N do
            Φ[i] ← Φ[i] + k(x[i], y[j]) × m[j]
```

Figure 3: P2P outer algorithm

From now on, without loss of generality, we will focus only on the **P2P inner** kernel in the case of the **electrostatic potential** (see (2)) in a simple two-dimensional application (*i.e*, $n = 2$). In other words, we aim to compute:

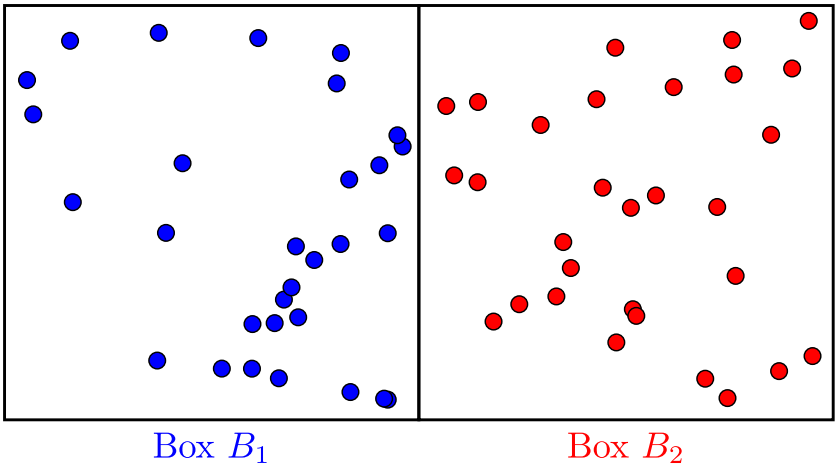

Figure 4: Bodies in neighboring boxes (typical situation for the P2P outer kernel)

$$p_i = \sum_{j=1}^{N} \frac{q_j}{\|x_i - x_j\|} \quad \forall i \in \{1, \ldots, N\} \qquad \text{with } x_i \in \mathbb{R}^2, \tag{4}$$

with p_i the potential of particle i and q_j the charge of particle j. This was the target kernel that we attempted to implement using Kokkos.

2.4 Kokkos in a nutshell

Before moving forward, let's provide a brief introduction to Kokkos [**9485033**, **carteredwards20143202**]. It is most of the time presented as a C++ library designed for **performance portability** in HPC applications. It allows developers to write parallel code that runs efficiently across **various hardware architectures**, including CPUs and GPUs, without the need to rewrite code for each platform. By providing abstractions for parallel execution and memory management, Kokkos allows users to leverage different backends (e.g., OpenMP, CUDA) through a single, unified API. However, it is important to note that these backends must be selected at compile-time. During the internship, we tested the following backends: **OpenMP (CPU)**, **C++ threads (CPU)**, and **CUDA (GPU)**. However, other backends are currently supported (such as **HIP**) and more are planned for the future. Last but not least, Kokkos relies on key abstractions, such as the memory space, which determines where the data is located, and the execution space,

which specifies where the code is executed.

The most convenient way to install Kokkos is via CMake (but you can find other alternatives in the documentation). However, Guix also offers an easy way to switch between different **backends**, making it simple to configure Kokkos for various architectures. For example, the following lines can be used to create development environments via guix shell using the `--pure` option:

- Using OpenMP as a backend:

```
guix shell --pure kokkos-openmp gcc-toolchain@11 <other
↪  packages> ...
```

- Using C++ threads as a backend:

```
guix shell --pure kokkos-threads gcc-toolchain@11 <other
↪  packages> ...
```

- Using CUDA as a backend:

```
guix shell --pure kokkos-cuda-a100 gcc-toolchain@11
↪  cuda-toolkit@12.4 <other packages> ...
export LD_PRELOAD=/usr/lib64/libcuda.so:/usr/lib64/libnvidia-p
↪  txjitcompiler.so # required to use CUDA with
↪  guix
```

We provide suitable **Guix environments** to run the implementations with these backends in the results section.

As mentioned earlier, Kokkos is written in modern C++ and makes extensive use of advanced C++ programming techniques. However, it offers a straightforward API, so users do not need to engage with complex C++ code to work with Kokkos. That said, it is still beneficial to be familiar with C++ **templates**. Roughly speaking, **Templates** allow for writing generic and reusable code that can handle different data types. They allow users to write **functions** and **classes** to be defined independently of specific types, so the same code can operate on various data types without duplication.

- A (non-exhaustive) starting point for getting a good grasp of C++ templates: Wikipedia - C++ Templates
- Entry point for templates in the reference documentation: cppreference.com - Templates