A Parallel Implementation of the Fast Fourier Transform Algorithm

Advanced Methods for Scientific Computing

Carrà Edoardo
edoardo.carra@mail.polimi.it

Gentile Lorenzo
lorenzo.gentile@mail.polimi.it

Ferrario Daniele
daniele.ferrario@mail.polimi.it

1 Introduction

The Fast Fourier Transform (FFT) is a powerful tool used in various fields, from pure mathematics to audio engineering and even finance. It's a method for expressing a function as a sum of periodic components, and for recovering the signal from those components. When both the function and its Fourier transform are replaced with discretized counterparts, it is called the discrete Fourier transform (DFT).

The FFT library we are introducing is a comprehensive and efficient implementation of the FFT algorithm. It was developed at MIT by Matteo Frigo and Steven G. Johnson, and it's known for its superior performance compared to other publicly available FFT software. The library's performance is portable, meaning the same program will perform well on most architectures without modification.

The FFT library is composed of several key components. The first part of the algorithm requires reordering the data through a bit-reversal procedure. This procedure can be viewed as a mapping with exclusive read and write access to the same position in the vector. This part of the execution is highly parallelizable, but it's not cache-friendly due to its non-contiguous memory access pattern.

2 Cooley-Tukey algorithm

Algorithm 1 Iterative FFT Algorithm

```
1: function FFT($a$)
2:     $bit\_reverse(a)$
3:     $n \leftarrow$ length($a$) for $s = 1$ to log($n$) do
4:     $m \leftarrow 2^s$
5:     $\omega_m \leftarrow \exp(-2\pi i/m)$ for $k = 0$ to $n - 1$ by $m$ do
6:         $\omega \leftarrow 1$ for $j = 0$ to $m/2 - 1$ do
7:             $t \leftarrow \omega \cdot a[k + j + m/2]$
8:             $u \leftarrow a[k + j]$
9:             $a[k + j] \leftarrow u + t$
10:            $a[k + j + m/2] \leftarrow u - t$
11:            $\omega \leftarrow \omega \cdot \omega_m$
           end
       end
       end
12:    return $a$ =0
```

3 Analysis of Cooley-Tukey algorithm

3.1 Complexity

The Cooley-Tukey algorithm has a time complexity of $O(NlogN)$. This is because the algorithm recursively breaks down a DFT of any composite size into many smaller DFTs, and this process is repeated log N times. At each stage, the algorithm performs N multiplications, leading to a total of N log N multiplications. Note that, in the beginning of the algorithm it is required to perform a bit-reversal operation which requires linear time to be performed. However, this is dominated by the $O(NlogN)$ time complexity of the FFT part of the algorithm.

3.2 Qualitative analysis

3.2.1 Partitioning

By examining the Cooley-Tukey algorithm, we can construct a dependency graph to identify potential task partitions for parallelization.

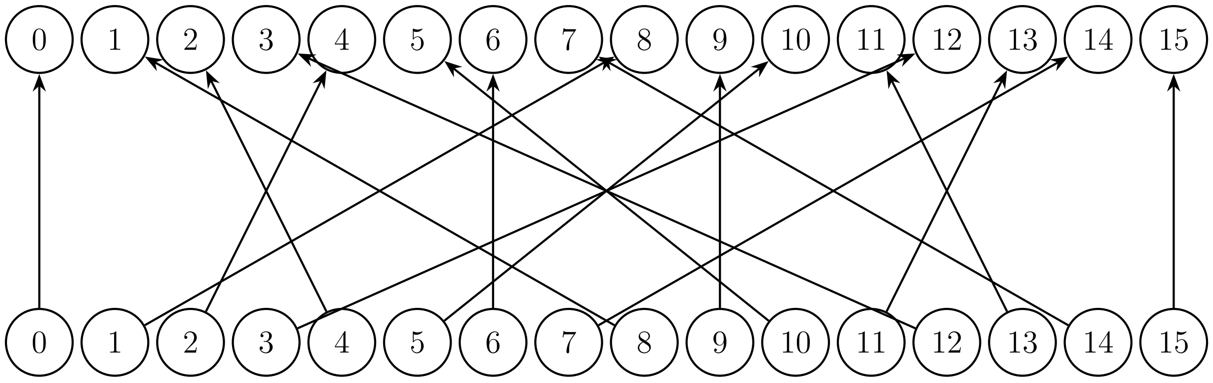


Figure 1: Example of bit reversal access with n=16

The initial stage of the algorithm involves reordering the data through a bit-reversal procedure. This procedure can be viewed as a mapping with exclusive read and write access to the same position in the vector. Please note that this part of the execution is highly parallelizable, but it's not cache-friendly due to its non-contiguous memory access pattern

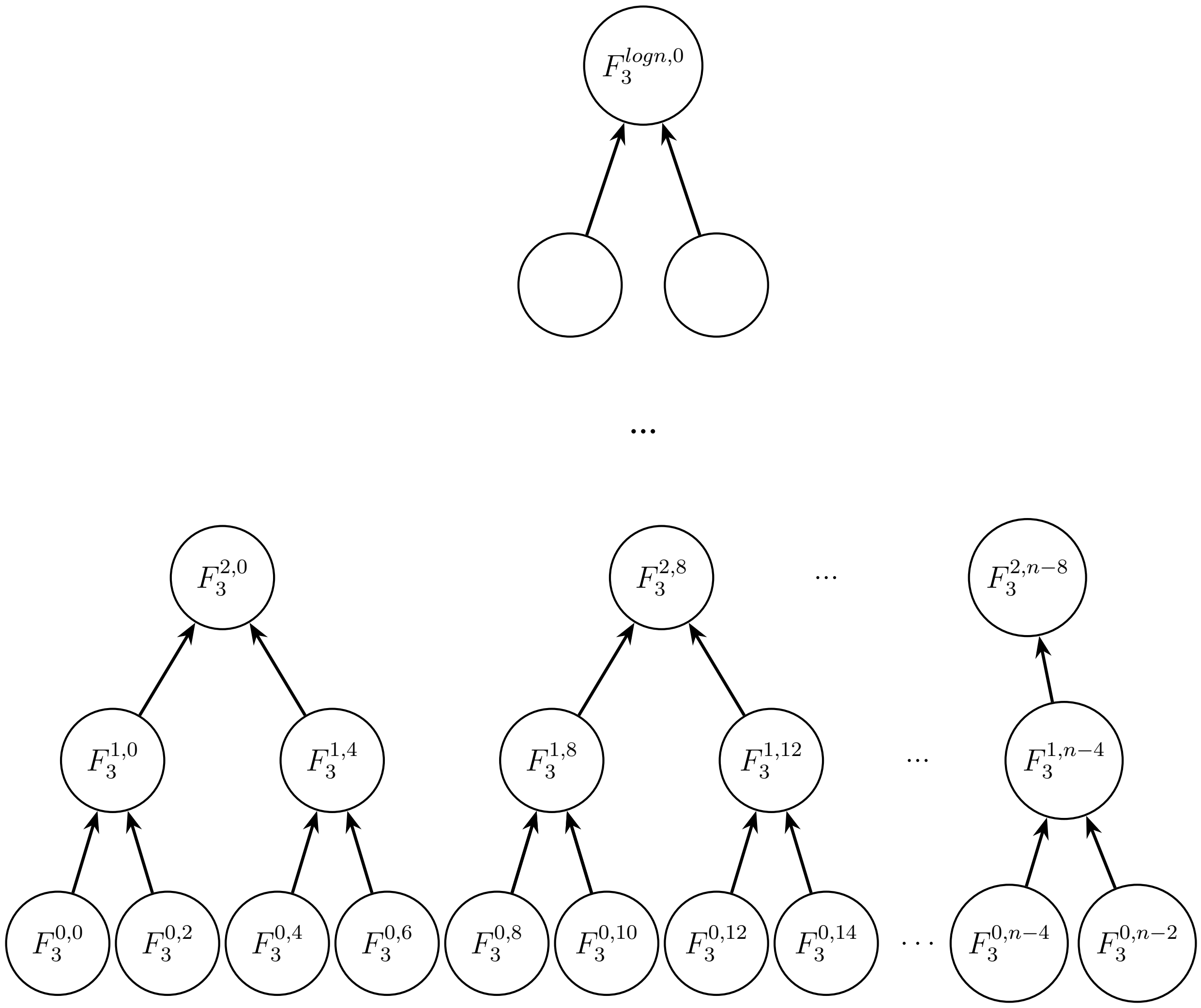
If we do not consider the bit reversal part, the algorithm is composed of three nested for-loops. To address them in the following paragraph, the following notation will be used:

1. F_1: the outer loop that executes $log(n)$ iterations. It's responsible for the main iteration over the data set and each iteration is called **stage**.
2. F_2^s: the first loop nested inside F_1 at the iteration s. It executes $n/2^s$ iterations, with k that is increased by 2^s at each iteration. This loop creates partitions of the input vector.
3. $F_2^{s,k}$: the inner loop at the iteration k of the loop F_2^s. It executes 2^{s-1} iterations. This loop performs the actual computation on each partition with a **butterfly** access to the input vector.

Upon initial inspection, it appears that a *lexically forward flow dependence* exists between each F_2^s loop in the Cooley-Tukey algorithm. This means that the execution of each F_2^s loop depends on the completion of the previous F2s loop.
To address this flow dependence in the algorithm, we can construct a dependency graph:

$$\begin{tikzcd} F_2^0 \arrow[r] & F_2^1 \arrow[r] & F_2^2 \arrow[r] & F_2^3 \arrow[r] & \cdots \arrow[r] & F_2^{logn} \end{tikzcd}$$

At each stage, $F_2^{s,k}$ are all independent. Furthermore, when we examine the dependency graph of these operations, we discover that a partition of the dependencies exists between each stage. This means that the tasks can be grouped in a way that allows for parallel execution:



Each $F_2^{s,k}$ performs a butterfly access of a partition of the vector, of length 2^s, as it shown in the following example:

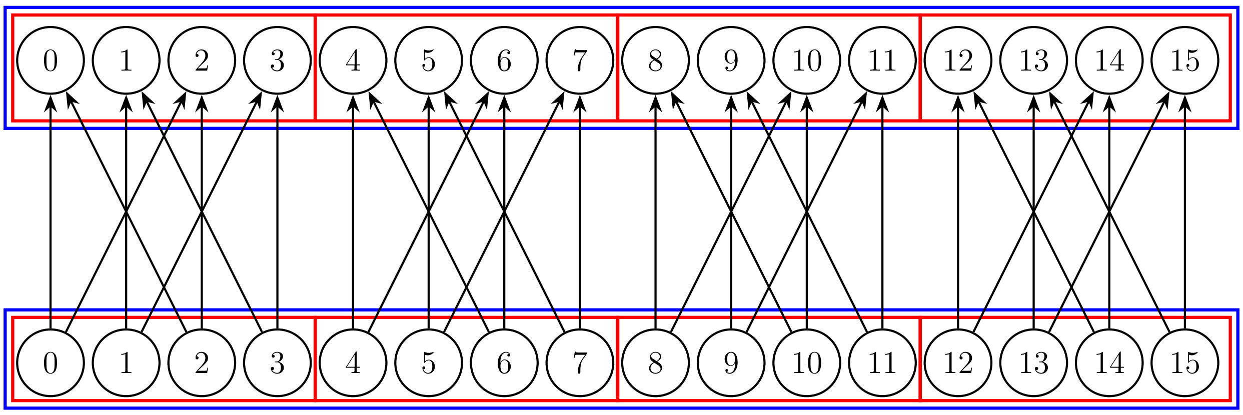
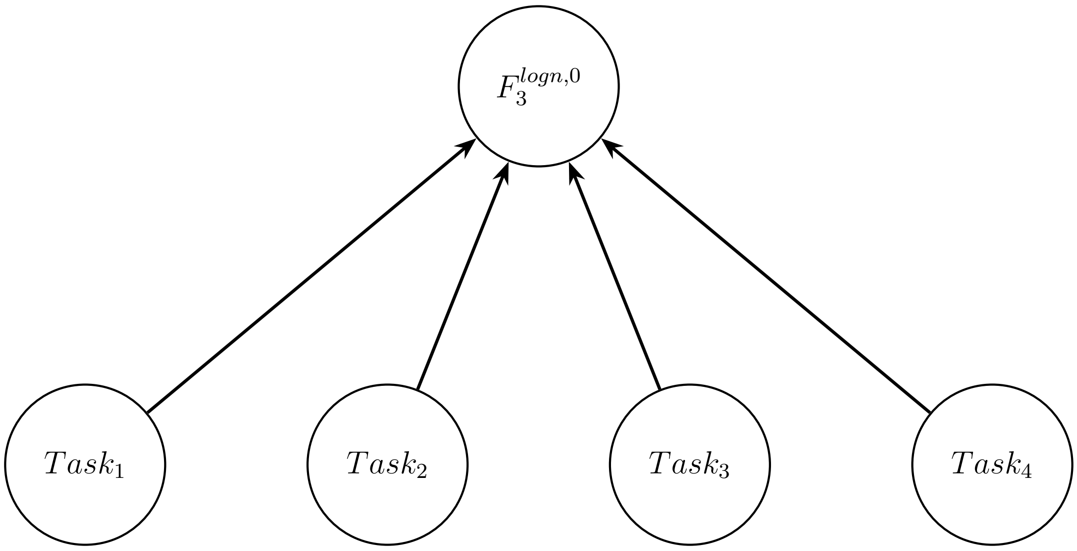


Figure 2: Example with n=16 of butterfly access at the second stage of the Cooley-Tukey algorithm

3.2.2 Communication

3.2.3 Aggregation



3.2.4 Mapping

4 Implementation

4.1 Classes and package structure

pattern strategy, principi di design: ISP IDP.

perché certe scelte?

4.2 Optimization

5 Tests and conclusion

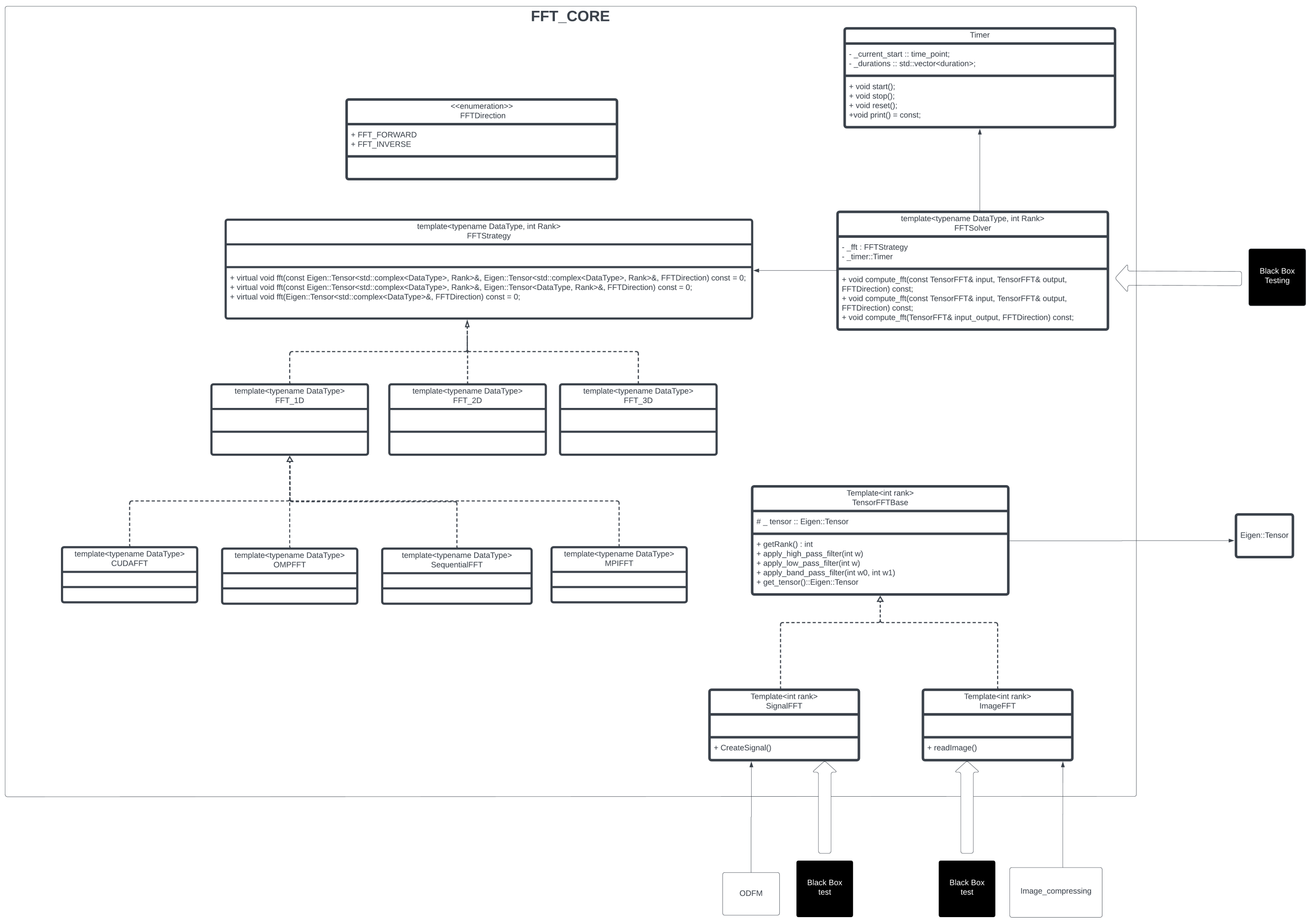

FFT_CORE
Timer
- _current_start :: time_point;
- _durations :: std::vector<duration>;
+ void start();
+ void stop();
+ void reset();
+void print() = const;
<<enumeration>>
FFTDirection
+ FFT_FORWARD
+ FFT_INVERSE
template<typename DataType, int Rank>
FFTStrategy
+ virtual void fft(const Eigen::Tensor<std::complex<DataType>, Rank>&, Eigen::Tensor<std::complex<DataType>, Rank>&, FFTDirection) const = 0;
+ virtual void fft(const Eigen::Tensor<std::complex<DataType>, Rank>&, Eigen::Tensor<DataType, Rank>&, FFTDirection) const = 0;
+ virtual void fft(Eigen::Tensor<std::complex<DataType>&, FFTDirection) const = 0;
template<typename DataType, int Rank>
FFTSolver
- _fft : FFTStrategy
- _timer::Timer
+ void compute_fft(const TensorFFT& input, TensorFFT& output, FFTDirection) const;
+ void compute_fft(const TensorFFT& input, TensorFFT& output, FFTDirection) const;
+ void compute_fft(TensorFFT& input_output, FFTDirection) const;
Black Box Testing
template<typename DataType>
FFT_1D
template<typename DataType>
FFT_2D
template<typename DataType>
FFT_3D
Template<int rank>
TensorFFTBase
# _tensor :: Eigen::Tensor
+ getRank() : int
+ apply_high_pass_filter(int w)
+ apply_low_pass_filter(int w)
+ apply_band_pass_filter(int w0, int w1)
+ get_tensor()::Eigen::Tensor
Eigen::Tensor
template<typename DataType>
CUDAFFT
template<typename DataType>
OMPFFT
template<typename DataType>
SequentialFFT
template<typename DataType>
MPIFFT
Template<int rank>
SignalFFT
+ CreateSignal()
Template<int rank>
ImageFFT
+ readImage()
ODFM
Black Box test
Black Box test
Image_compressing

Figure 3: Enter Caption