COMS4995 Parallel Functional Programming Project Proposal - FFT
Xuezhen Wang, Songheng Yin

1 Overview
The project will implement a Haskell program for the Fast Fourier Transform Algorithm. Specifically, we will provide two implementations for 1D and 2D signals. We will then apply parallel techniques to improve the performance on a large dataset.

2 Background
The discrete Fourier transform (DFT) is an important linear discrete transform that transforms a sequence of \( n \) complex numbers \( a_0, a_1, \ldots, a_{n-1} \), which can be represented in vector form \( a = (a_0, a_1, \ldots, a_{n-1})^T \), into another sequence \( c = (c_0, c_1, \ldots, c_{n-1})^T \) where

\[
c_k = \sum_{j=0}^{n-1} a_j \cdot e^{-\frac{2\pi i}{n}jk}
\]

where \( k = 0, 1, \ldots, n - 1 \). The direct computation trivially takes \( O(n^2) \) operations because it sums \( n \) terms for each output \( c_k \) out of a total of \( n \) outputs. Moreover, the inverse DFT defined as following has the same time complexity:

\[
a_j = \frac{1}{n} \sum_{k=0}^{n-1} c_k \cdot e^{\frac{2\pi i}{n}jk}
\]

for \( j = 0, 1, \ldots, n - 1 \)

The fast Fourier transform (FFT) reduces the complexity of computing DFT from \( O(n^2) \) to \( O(n \log n) \) using some nice properties of complex roots of unity and the divide-and-conquer technique, known as Cooley–Tukey algorithm. The FFT is a cornerstone in digital signal processing and computational mathematics, providing a powerful and efficient means to analyze the frequency content of discrete signals. Its applications span across diverse fields, underpinning numerous technologies and scientific advancements by enabling rapid and efficient frequency analysis.

3 Algorithm

Algorithm 1 Fast Fourier Transform (FFT)

1: procedure FFT(x)
2: \( N \leftarrow \text{length}(x) \)
3: if \( N = 1 \) then
4:     return x
5: end if
6: \( e \leftarrow \text{elements of } x \text{ at even indices} \)
7: \( o \leftarrow \text{elements of } x \text{ at odd indices} \)
8: \( E \leftarrow \text{FFT}(e) \)
9: \( O \leftarrow \text{FFT}(o) \)
10: Initialize vector \( X \) of length \( N \)
11: for \( k \leftarrow 0 \) to \( N/2 - 1 \) do
12:     \( t \leftarrow \exp(-2\pi i \cdot \frac{k}{N}) \)
13:     \( X[k] \leftarrow E[k] + t \cdot O[k] \)
14:     \( X[k + N/2] \leftarrow E[k] - t \cdot O[k] \)
15: end for
16: return X
17: end procedure

This is a basic recursive implementation pseudo-code for FFT. One distinct element which can be utilized in this implementation is we can parallel the algorithm into two sections for each recursion. The recursive way of FFT is easy to implement. However, it is easier to derive a parallel FFT algorithm when the sequential algorithm is in iterative form. Therefore, we also want to try the bottom-up version of the algorithm and compare the performance of these two.
After we finish the implementation of 1D FFT, we will implement a 2D FFT based on 1D FFT by processing the row sequentially first to get the intermediate 2D array and then processing each column on the intermediate 2D array.

4 Objectives and Workflow

4.1 Experiment Preparation
We will use a script to generate a large dataset including both random data and corner cases as the input to our Haskell program. The expected solution is generated simultaneously and can be used later to verify the correctness of our algorithm implementation.

4.2 Experiment Design
We are going to run the naive sequential and parallel algorithms respectively and monitor their performances. To ensure the effect of parallelism with GHC, we run with the \texttt{-threaded,\textbf{-O2}} options as well as \texttt{+RTS \textbf{-N}1 to \textbf{-N}8} on an 8-core machine.

4.3 Image Processing
We will utilize our 2D forward and backward FFT on various images and then display the frequency spectrum and the transformed back identity image.