Introduction

Numerical solutions of the acoustic wave equation play a critical role in the development of seismic imaging/inversion for hydrocarbon exploration. In the past decade, alternatives to finite difference (FD) schemes have emerged, such as the so-called Recursive integral time-extrapolation (RITE) methods (Du et al., 2013). These methods allow simulating accurate wave extrapolation with little numerical dispersion even when using larger time steps. However, all of them use mixed space/wavenumber-domain operators which are usually handled by FFT method, resulting in a great challenge when designing and implementing applications dealing with three-dimensional grids. There exists a number of FFT libraries written in different programming languages and providing different features. These libraries are optimized for supercomputers and scale well to large numbers of cores. Meanwhile, the absence of a common API makes it difficult to write applications that are able to use these libraries and to compare their performance. Developers consequently are faced with the tough choice of selecting a library before the code is implemented.

Motivated by this, Mohanan et al. (2019) have recently launched Fluidfft, an easy to use C++/Python API for performing FFTs with different libraries: FFTW (Frigo and Johnson, 2005), PFFT (Pippig, 2013), P3DFFT (Pekurovsky, 2012) and compute unified device architecture (CUDA) FFT library (cuFFT) from Nvidia. The package also includes utilities to benchmark the different FFT solutions for a particular case and on a particular machine. This feature, in particular, makes it attractive as the overall performance of the libraries can be easily obtained for different sizes of arrays and hardware. Such versatility allows applications build upon fluidfft to be efficient for different sizes and machines. To demonstrate the usefulness of fluidfft, we have implemented the one-step extrapolation matrix method (OSEM) proposed by Revelo and Pestana (2016) based on the fluidfft library. In those cases where various wavefield simulations are performed on the same domain (i.e. full-waveform inversion with uniform size shots), we could carry out only one simulation to find the library best suited to our needs before going ahead with the full simulation.

OSEM method for acoustic seismic modeling

The acoustic wave equation can be reformulated by introducing the analytical wavefield, $\hat{P}(x,t) = P(x,t) + iQ(x,t)$, where $x = \{x, y, z\}$. $Q(x,t) = H\{P(x,t)\}$ and $H\{\cdot\}$ is the Hilbert transform operator. For general media, the complex pressure wavefield satisfies the following first-order partial equation in time (Zhang and Zhang (2009))

$$\frac{\partial \hat{P}}{\partial t} + i\Phi \hat{P} = 0, \tag{1}$$

where $\Phi$ is a pseudodifferential operator in the space domain, defined by $\Phi = v\sqrt{-\nabla^2}$ and its symbol is $\phi = v(x)\sqrt{k_x^2 + k_y^2 + k_z^2}$, where $k_x$, $k_y$, and $k_z$ are the wavenumber components. Substituting the analytical wavefield into Eq. (1), we get the following coupled system of equations

$$\frac{\partial U}{\partial t} = \mathbf{A}U, \text{ with } \mathbf{A} = \begin{pmatrix} 0 & \Phi \\ -\Phi & 0 \end{pmatrix}, \tag{2}$$

where $U = [P, Q]^T$ is the $2(N_x \times N_y \times N_z)$ analytical pressure wavefield array, $\mathbf{A}$ is an antisymmetric matrix, and $\Phi$ is the pseudodifferential operator. Here, $N_x$, $N_y$, and $N_z$ denote the respective number of grid points in the $x$-, $y$- and $z$-directions.

The solution of Eq. (2) can be obtained by Tal-Ezer’s technique (Tal-Ezer et al. (1987)) and it is written as

$$\begin{pmatrix} P(t + \Delta t) \\ Q(t + \Delta t) \end{pmatrix} = \sum_{n=0}^{M} \varepsilon_n J_n(\Delta R) \tilde{T}(\frac{\mathbf{A}}{R}) \begin{pmatrix} P(t) \\ Q(t) \end{pmatrix} + \begin{pmatrix} f(t) \\ H\{f(t)\} \end{pmatrix}, \tag{3}$$

where $\varepsilon_0 = 1$, $\varepsilon_n = 2, n \geq 1$ and $J_n$ represents the Bessel function of order $n$, $\tilde{T}(z)$ are the modified Chebyshev polynomials, which satisfy the following recurrence relation: $\tilde{T}_{n+1}(z) = 2z \tilde{T}_n(z) + \tilde{T}_{n-1}(z)$, with $\tilde{T}_0(z) = I$ and $\tilde{T}_1(z) = z$. The right-hand terms $f(t)$ and $H\{f(t)\}$ are the source and its Hilbert
transform, respectively. The $R$ value is given by the maximum eigenvalue of $A$. For the 3D case $R = v_{\text{max}} \pi \sqrt{(1/\Delta x)^2 + (1/\Delta y)^2 + (1/\Delta z)^2}$, where $v_{\text{max}}$ is the maximum velocity of the model and $\Delta x, \Delta y$, and $\Delta z$ are the sizes of the grid along those dimensions. An $M$ value greater than $R \Delta t$ is required to ensure convergence. To implement the scheme in Eq. (3), the pseudodifferential operator $\Phi$ has to be computed. This is achieved by using forward ($\mathcal{F}$) and inverse ($\mathcal{F}^{-1}$) spatial Fourier transforms, as

$$\Phi[P] = v(x) \mathcal{F}^{-1} \{ \sqrt{k_x^2 + k_y^2 + k_z^2} \mathcal{F} \{ P \} \},$$

where $v(x)$ is the velocity and $k_x$, $k_y$, and $k_z$ are the spatial wavenumbers.

**Fluidfft**

Fluidfft provides a consistent collection of C++ classes and their Python wrapper classes for performing FFTs with different FFT libraries. Its main objective is to support research that relies on FFT, by facilitating the comparison and benchmarking of different FFT solutions in a consistent framework. Fluidfft was designed to be user-friendly while maintaining its focus on efficiency. The C++ API is implemented as hierarchy of classes according to Figure 1. The naming convention used for the classes <Type of FFT>With<Name of library> gives users an idea on how these work internally. By utilizing inheritance, the classes share the same function names and syntax defined in the base classes. Child classes take care of Fourier transformations based on the functionality of the specific library. In addition to FFT methods, other helper functions are provided by child classes.

In their work, Mohanan et al. (2019) provide a small piece of code as an example of how to use the C++ API (Figure 2). Switching between FFT libraries is as simple as changing the header file and class name (lines 3-4 and 14-15). Another important feature is that users can declare and allocate arrays without worrying about domain decomposition. Domain decomposition is handled by the constructor when an object of a FFT class is created (line 14). Furthermore, other member functions of the class take care of the allocation of data in real and spectral spaces (lines 16 and 17).

**Methodology and Results**

We performed scaling tests of our fluidfft-based OSEM code on YEMOJA Supercomputer at Senai-CIMATEC Supercomputing Center, which uses an InfiniBand interconnection. Each compute node used...
contains 128 GB of RAM and two sockets where each socket has an Intel Xeon E5-2680 v2 CPU at 2.8 GHz. Fluidfft was buildt using the Intel Compiler suite version 16 and Python 3.6.5. We made use of a 3D synthetic homogeneous isotropic model (\(v = 2500\) m/s) to benchmark the FFT classes. The model was discretized on a 320 \times 320 \times 320 mesh. The spatial sampling was 10 m in all directions. A Ricker wavelet with a center frequency of 30 Hz was used as the source. The observed data were simulated using such a model for a recording time of 1 s with a 1 ms sampling interval (i.e., \(nt = 1000\)). The latter could have been higher as the OSE matrix method ensures the stability for an arbitrary choice of \(\Delta t\). The simulation contains only one shot located in the center of the model. We used the boundary condition described by Cerjan et al. (1985). A strip of 60 nodes along the boundaries of the mesh is used, leading to a new extended 440 \times 440 \times 440 mesh. From this parameter configuration, an \(M = ceiling(\frac{R\Delta t}{\Delta t}) = 2\) is obtained, but we changed it to a bigger one \((M = 6)\), giving a better approximation. Therefore, the number of calls to the functions that calculate both forward and inverse FFTs is \((M - 1) \times nt \times 2 = 10000\). We used three different classes for the numerical simulation: FFT3DMPIWithFFTW1D, FFT3DMPIWithFFTWMPI3D and FFT3DMPIWithPFFT. The only changes we introduced to the FFT classes were the addition of the spatial sampling rates as constructor parameters and a function to compute the spatial wavenumbers so that they are created when an object is set up. To demonstrate the proper functioning of Fluidfft, we display the simulated snapshots computed by OSEM at the \(xz\)-plane \(y = 2.2\) km in Figure 3. In accordance with our expectations, the simulated snapshots by all classes are consistent.

The computational performance of the software was quantified using the speedup metric (Mohanan et al., 2019):

\[ S_\alpha = \frac{[\text{Time elapsed for } N \text{ iterations with } n_{p,\text{min}} \text{ processes}])_{\text{fastest}} \times n_{p,\text{min}}}{[\text{Time elapsed for } N \text{ iterations with } n_p \text{ processes}])_{\alpha}} \]

where \(n_{p,\text{min}}\) is the minimum number of processes employed for a specific array size and hardware. The subscripts, \(\alpha\) denotes the FFT class used and “fastest” corresponds to the fastest result among various FFT classes.

![Figure 3](image-url)

**Figure 3** Comparison of computed snapshots computed by the OSE matrix method using different FFT classes from FluidFFT in a homogeneous model. The time corresponds to 0.3 s of propagation. All the snapshots are shown in the same amplitude range.

We performed our scaling tests (on an array of size 440\(^3\)) using cores ranging from 2 to 220, as the latter is the largest divisor of 440 (not equal itself). We present the strong scaling result in Figure 4. The codes were implemented using double-precision and compiled without OpenMP support, due to the fact that PDFFT library does not support hybrid parallelism (MPI + OpenMP). Accordingly, the MPI processes were mapped to the cores. The results show that those classes that implement a 1D decomposition achieve the best performance. In our specific case, the fastest class was FFT3DMPIWithFFTWMPI3D. It may further be observed that the strong scaling, in general, exhibits sublinear behavior.

**Conclusions**

Fluidfft has been designed to be used by researchers in any area of study that involves numerical computations where real-to-complex/complex-to-real 2D or 3D FFTs are required. The code is implemented in C++ and can be used in a straightforward manner in the development of other C++ codes. It also includes Python wrappers, enabling it to be used in the development of Python projects as well. As an example,
Figure 4 Performance summary to solve $2 \times (M - 1) \times nt$ double-precision forward and inverse FFTs using fluidfft.

we used it to perform 3D acoustic seismic modeling through the OSEM method. This has enabled us to exploit available implementations of FFT, in order to determine which library best fits our application, without the need to reimplement algorithms from scratch. The benchmark results for a simple homogeneous model on the YEMOJA Supercomputer showed that the FFT3DMPIWithFFTWMPI3D class provides the best scaling for the considered size problem.

Acknowledgements

This work was supported by CNPq, CAPES, and INCT-GP/CNPq. The facility support from SENAI-CIMATEC Supercomputing Center is also acknowledged.

References


