

# HIGH-PERFORMANCE COMPUTING IN APPLIED MATHEMATICS

## TWO-DIMENSIONAL DISCRETE FOURIER TRANSFORM USING FAST FOURIER TRANSFORM

The 2D discrete Fourier transform is defined for a matrix  $a \in \mathbb{C}^{m \times n}$ .

*Definition:* Given a matrix  $a = (a_{i,j}) \in \mathbb{C}^{m \times n}$  we define the 2D Discrete Fourier Transform of  $a$  as the matrix  $y = (y_{l,k}) \in \mathbb{C}^{m \times n}$  whose entries are

$$y \stackrel{def}{=} DFT2(a), \quad y_{l,k} = \sum_{j=0}^{n-1} \sum_{q=0}^{m-1} a_{q,j} \omega_m^{lq} \omega_n^{jk}, \quad 0 \leq l \leq m-1, 0 \leq k \leq n-1 \quad (1)$$

where  $\omega_m$  and  $\omega_n$  are the principal roots of the unity of order  $m$  and  $n$ , respectively:

$$\omega_m = e^{2\pi i/m}, \quad \omega_n = e^{2\pi i/n}$$

If a standard approach is used, then evaluating  $DFT2(a)$  would require  $O(n^2 m^2)$  operations. Using the properties of DFT2 and one dimensional FFT the cost may be reduced to  $O(mn \log(mn))$  as we show next.

One notices that we may write (1) as

$$y_{l,k} = \sum_{j=0}^{n-1} \left[ \sum_{q=0}^{m-1} a_{q,j} \omega_m^{lq} \right] \omega_n^{jk}, \quad 0 \leq l \leq m-1, 0 \leq k \leq n-1 \quad (2)$$

For each fixed  $j$  in the range  $0 \leq j \leq n-1$ , the sum  $\sum_{q=0}^{m-1} a_{q,j} \omega_m^{lq}, 0 \leq l \leq m-1$  represents the one dimensional DFT of the column  $j$  of the matrix  $a$ ,  $a(:, j)$ . Thus if we denote

$$\hat{y}(:, j) = DFT(a(:, j)) \in \mathbb{C}^m, \quad \hat{y}(l, j) = \sum_{q=0}^{m-1} a_{q,j} \omega_m^{lq}, \quad 0 \leq l \leq m-1 \quad (3)$$

then  $DFT2(a)$  may be obtained as

$$y_{l,k} = \sum_{j=0}^{n-1} \hat{y}(l, j) \omega_n^{jk}, \quad 0 \leq l \leq m-1, 0 \leq k \leq n-1 \quad (4)$$

For each fixed  $l$  in the range  $0 \leq l \leq m-1$  the computation (4) represents the 1D DFT of the data  $\hat{y}(l, :) \in \mathbb{C}^n$ .

The computation (3) is performed using  $n$  1D FFTs for data of size  $m$ , at a computational cost  $cost_1 = nO(m \log m)$ . The computation of (4) is performed using an additional  $m$  1D FFTs of data of size  $n$ , at a computational cost  $cost_2 = mO(n \log n)$ . Overall,

$$cost = cost_1 + cost_2 = O(mn \log mn)$$

The algorithm for the 2D DFT using 1D FFT is

```
function fft2(a,m,n)

for j = 0 : n - 1
    haty(:,j) = fft(a(:,j),m)
end

for i = 0 : m - 1
    y(i,:) = fft(haty(i,:),n)
end
```

### *Parallel implementation P\_FFT2*

For the parallel implementation we assume that the number of processors is a divisor of both  $m$  and  $n$ , and define  $loc\_n = n/p$  and  $loc\_m = m/p$ .

The first loop in the *fft2* algorithm is over the columns of  $a$  (alternatively, you may reformulate it over the rows) and we may scatter  $a$  among processes, such that each process receives  $loc\_n$  columns in the local variable  $loc\_a$

$$loc\_a = a(:,j1 : j2), \quad \text{where } j1 = my\_rank * loc\_n, \quad j2 = (my\_rank + 1) * loc\_n - 1$$

and performs the loop of 1D FFTs

```
for j = 0 : loc_n - 1
    loc_haty(:,j) = fft(loc_a(:,j),m)
end
```

After the for loop is completed communication is required to obtain the proper rows of *haty* in each processor. A simple (but not efficient) solution is to gather *loc\_haty* into *haty* in all processes using *MPI\_Allgather*

The second loop of the *fft2* algorithm is then executed in parallel by distributing the computation over the rows

```
for i = 0 : loc_m - 1
    i1 = my_rank * loc_m + i
    loc_y(i,:) = fft(haty(i1,:),n)
end
```

After the loop is completed, process zero gathers *loc\_y* into  $y = P\_FFT2(a)$ . See *p\_fft2.f*