

## Topic 4: Multigrid Methods (continued)

Writing a clean program to parallelize the multigrid algorithm is tricky! The following problems must be solved:

- The problem grid must be partitioned among the processes.
- Each process must store extra information in “ghost layers” to implement
  - the smoothing algorithm,
  - the restriction algorithm, and
  - the prolongation algorithm.
- Ghost layer information needs to be updated synchronously prior to application of each of the above three algorithms.
- Updating ghost layers involves *a lot* of communication, but provides the opportunity to use *non-blocking* message passing to overlap communication and computation.
- In 2-D, each process has at most 4 neighbors with which it shares a side (North, South, East, West), and at most 4 neighbors with which it shares a vertex (NE, NW, SE, SW).

- As the grid is coarsened, some processes may end up with no grid points, which implies that the 8 neighbors of a process can change.

The program to implement all of this will be discussed next lecture.

## Fourier Transforms

Meanwhile, let's start discussing the FFT approach to solving elliptic PDE's. This algorithm scales like  $\mathcal{O}(N \log_2 N)$  where  $N$  is the number of discretization grid points. It is considerably more efficient than SOR, which scales like  $\mathcal{O}(N^{3/2})$ , and almost as efficient as Multigrid, which scales like  $\mathcal{O}(N)$ . The FFT has many applications, and is also an interesting algorithm to parallelize.

The basic reason why the Fourier Transform is useful in solving a PDE is that it converts a differential equation into an algebraic equation. Recall the definition of the Fourier transform  $g(k)$  of a function  $f(x)$ :

$$g(k) = \int_{-\infty}^{\infty} dx f(x) e^{-ikx} ,$$
$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} dk g(k) e^{ikx} .$$

The Fourier transform of the equation

$$\frac{\partial^2 \phi(x)}{\partial x^2} = -f(x) ,$$

can be Fourier transformed to

$$k^2 \chi(k) = -g(k) ,$$

where  $\chi(k)$  is the Fourier transform of  $\phi(x)$ .

## The Discrete Fourier Transform

The Discrete Fourier Transform or DFT is usually introduced in connection with a *time series*. Experimental measurement, or computer generation, of a function  $f(t)$  generally yields values of the function at a discrete set of values of the variable  $t$ :

$$f_n \equiv f(n \delta t) , \quad n = 0, 1, \dots, N-1 .$$

The function is measured, or sampled, at equally spaced time intervals of size  $\delta t$ . The quantity  $1/\delta t$  is called the *sampling rate*.

The  $N$  time series values can be expressed in terms of  $N$  harmonic components with *frequencies*

$$\omega_k = \frac{2k}{N} \omega_c , \quad k = 0, \dots, N-1 ,$$

where  $\omega_c = \pi/\delta t$  is called the Nyquist critical frequency. The DFT is defined by

$$g_k = \sum_{n=0}^{N-1} W^{kn} f_n , \quad W \equiv e^{i2\pi/N} .$$

Note that  $W$  is an  $N$ -th root of unity.

The time series can be reconstructed using the inverse DFT

$$f_n = \frac{1}{N} \sum_{k=0}^{N-1} W^{-nk} g_k .$$

Let's consider  $N = 2$ : here  $W = e^{i2\pi/2} = -1$ , so

$$\begin{aligned} f_0 &= \frac{1}{2} (g_0 + g_1) \\ f_1 &= \frac{1}{2} (g_0 - g_1) , \end{aligned}$$

which can be inverted to obtain the Fourier coefficients

$$\begin{aligned} g_0 &= (f_0 + f_1) \\ g_1 &= (f_0 - f_1) . \end{aligned}$$

One more example for  $N = 4$  and  $W = e^{i2\pi/4} = i$ . The discrete Fourier coefficients are given by:

$$\begin{aligned} g_0 &= i^0 f_0 + i^0 f_1 + i^0 f_2 + i^0 f_3 = f_0 + f_1 + f_2 + f_3 \\ g_1 &= i^0 f_0 + i^1 f_1 + i^2 f_2 + i^3 f_3 = f_0 + i f_1 - f_2 - i f_3 \\ g_2 &= i^0 f_0 + i^2 f_1 + i^4 f_2 + i^6 f_3 = f_0 - f_1 + f_2 - f_3 \\ g_3 &= i^0 f_0 + i^3 f_1 + i^6 f_2 + i^9 f_3 = f_0 - i f_1 - f_2 + i f_3 . \end{aligned}$$

The DFT is given by:

$$\begin{aligned}
 f_0 &= \frac{1}{4} (g_0 + g_1 + g_2 + g_3) \\
 f_1 &= \frac{1}{4} (g_0 - ig_1 - g_2 + ig_3) \\
 f_2 &= \frac{1}{4} (g_0 - g_1 + g_2 - g_3) \\
 f_3 &= \frac{1}{4} (g_0 + ig_1 - g_2 - ig_3) .
 \end{aligned}$$

The formulas for the DFT and its inverse are therefore extremely simple, but evaluating them numerically can be problematic for large  $N$ .

Computing such a discrete Fourier transform involves  $\mathcal{O}(N^2)$  operations, because there are  $N - 1$  coefficients each represented as a sum of  $N - 1$  terms. The FFT (Fast Fourier Transform) algorithm computes the transform in  $\mathcal{O}(N \log_2 N)$  operations. This is a significant speedup for large data sets:

N	CPU Time Required at $10^6$ Flops	
	Discrete Fourier Transform	Fast Fourier Transform
$10^3$	1.0 sec	0.01 sec
$10^6$	$10^6$ sec = 12 days	20 sec
$10^9$	$10^{12}$ sec = 32,000 years	$3.0 \times 10^5$ sec = 8.3 hours

## The Fast Fourier Transform

Cooley and Tuckey found a very clever algorithm for reducing the number of operations to  $\mathcal{O}(N \log_2 N)$ . One reason why this is possible is that the  $N^2$  elements  $W^{kj}$  are not all distinct: in fact there are only  $N$  different values. The second essential ingredient in this algorithm is the assumption that  $N$  is a power of 2: this makes possible a *divide and conquer* strategy in which  $N$  is halved in each step; there are  $\log_2 N$  such steps, and each involves  $\mathcal{O}(N)$  operations which implement a lemma due to Danielson and Lanczos.

Let's examine the workings of the FFT algorithm for the case  $N = 4$  with  $W = \exp(i\pi/2) = i$ . The algorithm first *bit-reverses* the  $f_n$  values, and then takes  $\log_2 4 = 2$  Danielson-Lanczos steps:

$$\begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{pmatrix} \rightarrow \begin{pmatrix} f_0 \\ f_2 \\ f_1 \\ f_3 \end{pmatrix} \rightarrow \begin{pmatrix} f_0 + W^0 * f_2 \\ f_0 - W^0 * f_2 \\ f_1 + W^0 * f_3 \\ f_1 - W^0 * f_3 \end{pmatrix} \rightarrow \begin{pmatrix} [f_0 + W^0 f_2] + W^0 * [f_1 + W^0 f_3] \\ [f_0 - W^0 f_2] + W^1 * [f_1 - W^0 f_3] \\ [f_0 + W^0 f_2] - W^0 * [f_1 + W^0 f_3] \\ [f_0 - W^0 f_2] - W^1 * [f_1 - W^0 f_3] \end{pmatrix} = \begin{pmatrix} g_0 \\ g_1 \\ g_2 \\ g_3 \end{pmatrix}.$$

Note that the expression in braces [ ] do not need to be recalculated! The whole calculation therefore involves only  $N \log_2 N = 4 \times 2 = 8$  complex multiplications (\*) and additions ( $\pm$ ) or subtractions! This compares with  $N^2 = 4^2 = 16$  for straightforward matrix multiplication:

$$\begin{pmatrix} g_0 \\ g_1 \\ g_2 \\ g_3 \end{pmatrix} = \begin{pmatrix} W^0 * f_0 + W^0 * f_1 + W^0 * f_2 + W^0 * f_3 \\ W^0 * f_0 + W^1 * f_1 + W^2 * f_2 + W^3 * f_3 \\ W^0 * f_0 + W^2 * f_1 + W^4 * f_2 + W^6 * f_3 \\ W^0 * f_0 + W^3 * f_1 + W^6 * f_2 + W^9 * f_3 \end{pmatrix} = \begin{pmatrix} f_0 + f_1 + f_2 + f_3 \\ f_0 - if_1 - f_2 + if_3 \\ f_0 - f_1 + f_2 - f_3 \\ f_0 + if_1 - f_2 - if_3 \end{pmatrix}.$$

Note also that the algorithm transforms  $f$  into  $g$  *in place*!

### Bit Reversal Magic

The first step in the FFT algorithm is to rearrange the  $f$  values by reversing the bits in the component indices:

$$\begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} f_{00} \\ f_{01} \\ f_{10} \\ f_{11} \end{pmatrix} \rightarrow \begin{pmatrix} f_{00} \\ f_{10} \\ f_{01} \\ f_{11} \end{pmatrix} = \begin{pmatrix} f_0 \\ f_2 \\ f_1 \\ f_3 \end{pmatrix}$$

One more example of bit reversal with  $N = 8 = 2^3$  and  $W = \exp(i\pi/4)$ :

$$\begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \end{pmatrix} = \begin{pmatrix} f_{000} \\ f_{001} \\ f_{010} \\ f_{011} \\ f_{100} \\ f_{101} \\ f_{110} \\ f_{111} \end{pmatrix} \rightarrow \begin{pmatrix} f_{000} \\ f_{100} \\ f_{010} \\ f_{110} \\ f_{001} \\ f_{101} \\ f_{011} \\ f_{111} \end{pmatrix} = \begin{pmatrix} f_0 \\ f_4 \\ f_2 \\ f_6 \\ f_1 \\ f_5 \\ f_3 \\ f_7 \end{pmatrix}$$

$$\begin{pmatrix} f_0 + W^0 * f_4 \\ f_0 - W^0 * f_4 \\ f_2 + W^0 * f_6 \\ f_2 - W^0 * f_6 \\ f_1 + W^0 * f_5 \\ f_1 - W^0 * f_5 \\ f_3 + W^0 * f_7 \\ f_3 - W^0 * f_7 \end{pmatrix} \rightarrow \begin{pmatrix} [f_0 + W^0 f_4] + W^0 * [f_2 + W^0 f_6] \\ [f_0 - W^0 f_4] + W^2 * [f_2 - W^0 f_6] \\ [f_0 + W^0 f_4] - W^0 * [f_2 + W^0 f_6] \\ [f_0 - W^0 f_4] - W^2 * [f_2 - W^0 f_6] \\ [f_1 + W^0 f_5] + W^0 * [f_3 + W^0 f_7] \\ [f_1 - W^0 f_5] + W^2 * [f_3 - W^0 f_7] \\ [f_1 + W^0 f_5] - W^0 * [f_3 + W^0 f_7] \\ [f_1 - W^0 f_5] - W^2 * [f_3 - W^0 f_7] \end{pmatrix}$$

and a third step yields

$$\rightarrow \begin{pmatrix} [[f_0 + W^0 f_4] + W^0 [f_2 + W^0 f_6]] + W^0 * [[f_1 + W^0 f_5] + W^0 [f_3 + W^0 f_7]] \\ [[f_0 - W^0 f_4] + W^2 [f_2 - W^0 f_6]] + W^1 * [[f_1 - W^0 f_5] + W^2 [f_3 - W^0 f_7]] \\ [[f_0 + W^0 f_4] - W^0 [f_2 + W^0 f_6]] + W^2 * [[f_1 + W^0 f_5] - W^0 [f_3 + W^0 f_7]] \\ [[f_0 - W^0 f_4] - W^2 [f_2 - W^0 f_6]] + W^3 * [[f_1 - W^0 f_5] - W^2 [f_3 - W^0 f_7]] \\ [[f_0 + W^0 f_4] + W^0 [f_2 + W^0 f_6]] - W^0 * [[f_1 + W^0 f_5] + W^0 [f_3 + W^0 f_7]] \\ [[f_0 - W^0 f_4] + W^2 [f_2 - W^0 f_6]] - W^1 * [[f_1 - W^0 f_5] + W^2 [f_3 - W^0 f_7]] \\ [[f_0 + W^0 f_4] - W^0 [f_2 + W^0 f_6]] - W^2 * [[f_1 + W^0 f_5] - W^0 [f_3 + W^0 f_7]] \\ [[f_0 - W^0 f_4] - W^2 [f_2 - W^0 f_6]] - W^3 * [[f_1 - W^0 f_5] - W^2 [f_3 - W^0 f_7]] \end{pmatrix}.$$

The number of complex multiplications (\*) required is  $N = 8$  in each of the  $\log_2 N = \log_2 8 = 3$  steps for a total of  $3 \times 8 = 24$  compared with  $N^2 = 8^2 = 64$  for matrix multiplication.

### *Danielson-Lanczos Lemma*

Note that bit-reversal separates the even indices from the odd indices. As explained

in *Numerical Recipes*, the Danielson-Lanczos lemma

$$F_k = \sum_{j=0}^{N-1} W^{kj} f_j = \sum_{j=0}^{N/2-1} (W^2)^{kj} f_{2j} + W^k \sum_{j=0}^{N/2-1} (W^2)^{kj} f_{2j+1}$$

decomposes the sum into even and odd terms, each of which is a discrete Fourier transform of length  $N/2$ : this lends itself to the divide and conquer algorithm illustrated in the examples above. The algorithm can also exploit the following symmetry

$$W^{k+\frac{N}{2}} = e^{-i(2\pi/N)(k+\frac{N}{2})} = e^{-i(2\pi/N)k} e^{-i\pi} = -W^k,$$

of the  $N$ -th roots of unity for  $N$  even to halve the number of powers of  $W$  required. For the case  $N = 8$ , the FFT algorithm divide and conquer strategy starts with 4 transforms of length 2 with  $W_2 = W^4 = -1$

$$\begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \end{pmatrix} \rightarrow \begin{pmatrix} f_0 \\ f_4 \\ f_2 \\ f_6 \\ f_1 \\ f_5 \\ f_3 \\ f_7 \end{pmatrix} \rightarrow \begin{pmatrix} f_0 + W^0 * f_4 \\ f_0 - W^0 * f_4 \\ f_2 + W^0 * f_6 \\ f_2 - W^0 * f_6 \\ f_1 + W^0 * f_5 \\ f_1 - W^0 * f_5 \\ f_3 + W^0 * f_7 \\ f_3 - W^0 * f_7 \end{pmatrix} \equiv \begin{pmatrix} f'_0 \\ f'_4 \\ f'_2 \\ f'_6 \\ f'_1 \\ f'_5 \\ f'_3 \\ f'_7 \end{pmatrix}$$

followed by 2 transforms of length 4 with  $W_4 = W^2 = -i$ , and a single transform of

length 8

$$\begin{pmatrix} f'_0 \\ f'_4 \\ f'_2 \\ f'_6 \\ f'_1 \\ f'_5 \\ f'_3 \\ f'_7 \end{pmatrix} \rightarrow \begin{pmatrix} f'_0 + W^0 * f'_2 \\ f'_4 + W^2 * f'_6 \\ f'_0 - W^0 * f'_2 \\ f'_4 - W^2 * f'_6 \\ f'_1 + W^0 * f'_3 \\ f'_5 + W^2 * f'_7 \\ f'_1 - W^0 * f'_3 \\ f'_5 - W^2 * f'_7 \end{pmatrix} \equiv \begin{pmatrix} f''_0 \\ f''_4 \\ f''_2 \\ f''_6 \\ f''_1 \\ f''_5 \\ f''_3 \\ f''_7 \end{pmatrix} \rightarrow \begin{pmatrix} f''_0 + W^0 * f''_1 \\ f''_4 + W^1 * f''_5 \\ f''_2 + W^2 * f''_3 \\ f''_6 + W^3 * f''_7 \\ f''_0 - W^0 * f''_1 \\ f''_4 - W^1 * f''_5 \\ f''_2 - W^2 * f''_3 \\ f''_6 - W^3 * f''_7 \end{pmatrix} = \begin{pmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \\ F_7 \end{pmatrix} .$$

Note once again the  $N \log_2 N = 8 \cdot 3 = 24$  complex multiplications required to perform this transform.