Lecture 7

FFT

7.1 FFT

The *Fast Fourier Transform* is perhaps the most important subroutine in scientific computing. It has applications ranging from multiplying numbers and polynomials to image and signal processing, time series analysis, and the solution of linear systems and PDEs. There are tons of books on the subject including two recent worderful ones by Charles van Loand and Briggs.

The discrete Fourier transform of a vector x is $y = F_n x$, where F_n is the $n \times n$ matrix whose entry $(F_n)_{jk} = e^{-2\pi ijk/n}$, $j, k = 0 \ldots n-1$. It is nearly always a good idea to use 0 based notation (as with the C programming language) in the context of the discrete Fourier transform. The negative exponent corresponds to Matlab's definition. Indeed in matlab we obtain `fn=fft(eye(n))`.

A good example is

$$F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}.$$

Sometimes it is convenient to denote $(F_n)_{jk} = \omega_n^{jk}$, where $\omega_n = e^{-2\pi/n}$.

The Fourier matrix has more interesting properties than any matrix deserves to have. It is symmetric (but not Hermitian). It is Vandermonde (but not ill-conditioned). It is unitary except for a scale factor ($\frac{1}{\sqrt{n}} F_n$ is unitary). In two ways the matrix is connected to group characters: the matrix itself is the character table of the finite cyclic group, and the eigenvectors of the matrix are determined from the character table of a multiplicative group.

The trivial way to do the Fourier transform is to compute the matrix-vector multiply requiring n^2 multipilications and roughly the same number of additions. Cooley and Tukey gave the first $O(n \log n)$ time algorithm (actually the algorithm may be found in Gauss' work) known today as the FFT algorithm. We shall assume that $n = 2^p$.

The Fourier matrix has the simple property that if Π_n is an unshuffle operation, then

$$F_n \Pi_n^T = \begin{pmatrix} F_{n/2} & D_n F_{n/2} \\ F_{n/2} & -D_n F_{n/2} \end{pmatrix}, \tag{7.1}$$

where D_n is the diagonal matrix $\mathrm{diag}(1, \omega_n, \ldots, \omega_n^{n/2-1})$.

One DFT algorithm is then simply: 1) unshuffle the vector 2) recursively apply the FFT algorithm to the top half and the bottom half, then combine elements in the top part with corresponding elements in the bottom part ("the butterfly") as prescribed by the matrix $\begin{pmatrix} I & D_n \\ I & -D_n \end{pmatrix}$.

Everybody has their favorite way to visualize the FFT algorithm. For us, the right way is to think of the data as living on a hypercube. The algorithm is then, permute the cube, perform the FFT on a pair of opposite faces, and then perform the butterfly, along edges across the dimension connecting the opposite faces.

We now repeat the three steps of the recursive algorithm in index notation:

- Step 1: $i_{d-1} \dots i_1 i_0 \rightarrow i_0 i_{d-1} \dots i_1$
- Step 2: $i_0 i_{d-1} \dots i_1 \rightarrow i_0 \mathrm{fft}(i_{d-1} \dots i_1)$
- Step 3: $\rightarrow \bar{i_0} \mathrm{fft}(i_{d-1} \dots i_1)$

Here Step 1 is a data permutation, Step 2 refters to two FFTs, and Step 3 is the butterfly on the high order bit.

In conventional notation:

$$y_j = (F_n x)_j = \sum_{k=0}^{n-1} \omega_n^{jk} x_k$$

can be cut into the even and the odd parts:

$$y_j = \sum_{k=0}^{m-1} \omega_n^{2jk} x_{2k} + \omega_n^j \left(\sum_{k=0}^{m-1} \omega_n^{2jk} x_{2k+1} \right) ;$$

since $\omega_n^2 = \omega_m$, the two sums are just $\mathrm{FFT}(x_{\mathrm{even}})$ and $\mathrm{FFT}(x_{\mathrm{odd}})$. With this remark (see Fig. 1),

$$\begin{array}{rcl} y_j & = & \sum_{k=0}^{m-1} \omega_m^{jk} x_{2k} \quad + \quad \omega_n^j \left(\sum_{k=0}^{m-1} \omega_m^{jk} x_{2k+1} \right) \\ y_{j+m} & = & \sum_{k=0}^{m-1} \omega_m^{jk} x_{2k} \quad - \quad \omega_n^j \left(\sum_{k=0}^{m-1} \omega_m^{jk} x_{2k+1} \right) \end{array} .$$

Then the algorithm keeps recurring; the entire "communication" needed for an FFT on a vector of length 8 can be seen in Fig. 2

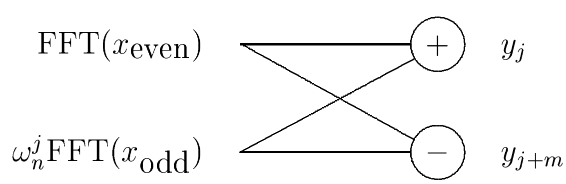

Figure 7.1: Recursive block of the FFT.

The number of operations for an FFT on a vector of length n equals to twice the number for an FFT on length $n/2$ plus $n/2$ on the top level. As the solution of this recurrence, we get that the total number of operations is $\frac{1}{2} n \log n$.

Now we analyze the data motion required to perform the FFT. First we assume that to each processor one element of the vector x is assigned. Later we discuss the "real-life" case when the number of processors is less than n and hence each processor has some subset of elements. We also discuss how FFT is implemented on the CM-2 and the CM-5.

The FFT always goes from high order bit to low order bit, i.e., there is a fundamental asymmetry that is not evident in the figures below. This seems to be related to the fact that one can obtain a subgroup of the cyclic group by alternating elements, but not by taking, say, the first half of the elements.

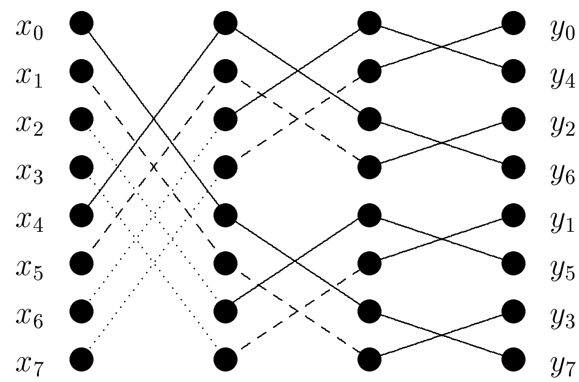

Figure 7.2: FFT network for 8 elements. (Such a network is not built in practice)

7.1.1 Data motion

Let $i_p i_{p-1} \dots i_2 i_1 i_0$ be a bit sequence. Let us call $i_0 i_1 i_2 \dots i_{p-1} i_p$ the **bit reversal** of this sequence. The important property of the FFT network is that if the i-th input is assigned to the i-th processor for $i \leq n$, then the i-th output is found at the processor with address *the bit-reverse of* i. Consequently, if the input is assigned to processors with bit-reversed order, then the output is in standard order. The inverse FFT reverses bits in the same way.

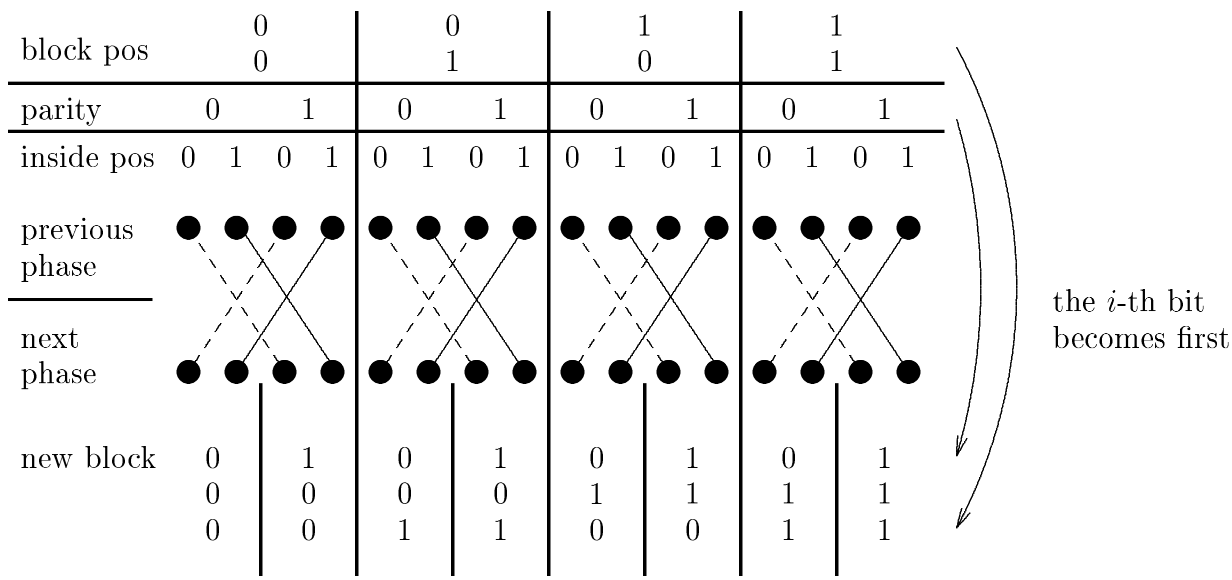

Figure 7.3: The output of FFT is in bit-reversed order.

To see why FFT reverses the bit order, let us have a look at the i-th segment of the FFT network (Fig. 3). The input is divided into parts and the current input (top side) consists of FFT's of these parts. One "block" of the input consists of the same fixed output element of all the parts. The $i-1$ most significant bits of the input address determine this output element, while the least significant bits the the part of the original input whose transforms are at this level.

The next step of the FFT computes the Fourier transform of twice larger parts; these consist

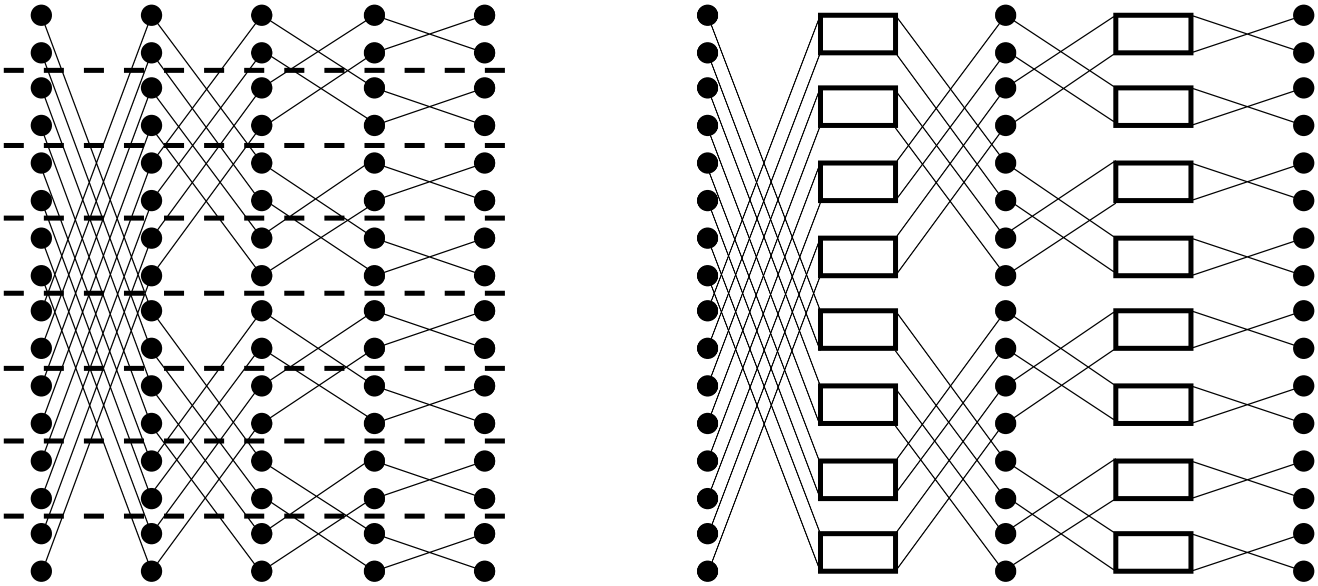

Figure 7.4: Left: more than one element per processor. Right: one box is a 4×4 matrix multiply.

of an "even" and an "odd" original part. Parity is determined by the i-th most significant bit.

Now let us have a look at one unit of the network in Fig. 1; the two inputs correspond to the same even and odd parts, while the two outputs are the possible "farthest" vector elements, they differ in the most significant bit. What happens is that the i-th bit jumps first and becomes most significant (see Fig. 3).

Now let us follow the data motion in the entire FFT network. Let us assume that the i-th input element is assigned to processor i. Then after the second step a processor with binary address $i_p i_{p-1} i_{p-2} \ldots i_1 i_0$ has the $i_{p-1} i_p i_{p-2} \ldots i_1 i_0$-th data, the second bit jumps first. Then the third, fourth, ..., p-th bits all jump first and finally that processor has the $i_0 i_1 i_2 \ldots i_{p-1} i_p$-th output element.

7.1.2 FFT on parallel machines

In a realistic case, on a parallel machine some bits in the input address are local to one processor. The communication network can be seen in Fig. 4, left. FFT requires a large amount of communication; indeed it has fewer operations per communication than usual dense linear algebra. One way to increase this ratio is to *combine some layers* into one, as in Fig. 4, right. If s consecutive layers are combined, in one step a $2^s \times 2^s$ matrix multiplication must be performed. Since matrix multiplication vectorizes and usually there are optimized routines to do it, such a step is more efficient than communicating all small parts. Such a modified algorithm is called the *High Radix* FFT.

The FFT algorithm on the CM-2 is basically a High Radix FFT. However, on the CM-5 data motion is organized in a different way. The idea is the following: if s bits of the address are local to one processor, the last s phases of the FFT do not require communication. Let 3 bits be local to one processor, say. On the CM-5 the following data rearrangement is made: the data from the

$$i_p i_{p-1} i_{p-2} \ldots i_3 | i_2 i_1 i_0 \text{ -th}$$

processor is moved to the

$$i_2 i_1 i_0 i_{p-3} i_{p-4} \ldots i_3 | i_p i_{p-1} i_{p-2} \text{ -th!}$$

This data motion can be arranged in a clever way; after that the next 3 steps are local to processors. Hence the idea is to perform all communication at once *before* the actual operations are made.

[Not yet written: The six step FFT]

[Not yet written: FFTW]

[Good idea: Use the picture in our paper first to illustrate the notation]

7.1.3 Exercises

1. Verify equation (**??**).

2. Just for fun, find out about the FFT through Matlab.

   We are big fans of the `phone` command for those students who do not already have a good physical feeling for taking Fourier transforms. This command shows (and plays if you have a speaker!) the signal generated when pressing a touch tone telephone in the United States and many other countries. In the old days, when a pushbutton phone was broken apart, you could see that pressing a key depressed one lever for an entire row and another lever for an entire column. (For example, pressing 4 would depress the lever corresponding to the second row and the lever corresponding to the first column.)

   To look at the FFT matrix, in a way, `plot(fft(eye(7)));axis('square')`.

3. In Matlab use the `flops` function to obtain a flops count for FFT's for different power of 2 size FFT's. Make you input complex. Guess a flop count of the form $a + bn + c \log n + dn \log n$. Remembering that Matlab's \ operator solves least squares problems, find a, b, c and d. Guess whether Matlab is counting flops or using a formula.

7.2 Matrix Multiplication

Everyone thinks that to multiply two 2-by-2 matrices requires 8 multiplications. However, Strassen gave a method, which requires only 7 multiplications! Actually, compared to the 8 multiplies and 4 adds of the traditional way, Strassen's method requires only 7 multiplies but 18 adds. Nowadays when multiplication of numbers is as fast as addition, this does not seem so important. However when we think of block matrices, matrix multiplication is very slow compared to addition. Strassen's method will give an $O(n^{2.8074})$ algorithm for matrix multiplication, in a recursive way very similar to the FFT.

First we describe Strassen's method for two block matrices:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} P_1 + P_4 - P_5 + P_7 & P_3 + P_5 \\ P_2 + P_4 & P_1 + P_3 - P_2 + P_6 \end{pmatrix}$$

where

$$\begin{aligned} P_1 &= (A_{1,1} + A_{1,2})(B_{1,1} + B_{2,2}) \ , \\ P_2 &= (A_{2,1} + A_{2,2})B_{1,1} \ , \\ P_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \ , \\ P_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \ , \\ P_5 &= (A_{1,1} + A_{1,2})B_{2,2} \ , \end{aligned}$$

$$\begin{aligned} P_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \ , \\ P_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \ . \end{aligned}$$

If, as in the FFT algorithm, we assume that $n = 2^p$, the matrix multiply of two n-by-n matrices calls 7 multiplications of $(n/2)$-by-$(n/2)$ matrices. Hence the time required for this algorithm is $O(n^{\log_2 7}) = O(n^{2.8074})$. Note that Strassen's idea can further be improved (of course, with the loss that several additions have to be made and the constant is impractically large) the current such record is an $O(n^{2.376})$-time algorithm.

A final note is that, again as in the FFT implementations, we do not recur and use Strassen's method with 2-by-2 matrices. For some sufficient p, we stop when we get $2^p \times 2^p$ matrices and use direct matrix multiply which vectorizes well on the machine.

7.3 Basic Data Communication Operations

We conclude this section by list the set of basic data communication operations that are commonly used in a parallel program.

- **Single Source Broadcast**:
- **All-to-All Broadcast**:
- **All-to-All Personalized Communication**:
- **Array Indexing or Permutation**: There are two types of array indexing: the *left* array indexing and the *right* array indexing.
- **Polyshift**: SHIFT and EOSHIFT.
- **Sparse Gather and Scatter**:
- **Reduction and Scan**: