

Lecture 20: April 9

Lecturer: David Bailey

Scribes: David Eitan Poll

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

20.1 Discrete Fourier Transforms (DFT)

A DFT, or Discrete Fourier Transform, is a means for transforming one function into another, and is often used in signal processing. Next, we will show how to compute a DFT.

20.1.1 Computing a DFT

N numbers x_0, \dots, x_{N-1} are transformed by DFT into the sequence of N numbers X_0, \dots, X_{N-1} by the following formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}$$

20.1.2 Fast Fourier Transform

The Fast Fourier Transform (FFT) is a "Fast" method of computing DFTs. To contrast their runtimes:

- Traditional DFT: $T(N) \approx 2n^2$
- FFT: $T(N) \approx 5n \log n$

The FFT of a function is often referred to simply as F . Furthermore, there is a relationship between an FFT and its inverse:

$$F^{-1} = \frac{1}{N} F^*$$

20.2 Poisson's Equation

Poisson's equation arises in many models for electrostatics, mechanical engineering, and theoretical physics. It is a partial differential equation that applies in any number of dimensions with broad use, and can be computed by FFT in $O(N \log N)$ time, where $N = n^2$ vars.

Poisson's equation is as follows:

- In 1-d: $\frac{d^2 x}{dx^2} = f(x)$

- In 2-d: $\frac{\partial^2 x}{\partial x^2} + \frac{\partial^2 y}{\partial y^2} = f(x, y)$
- In 3-d: $\frac{\partial^2 x}{\partial x^2} + \frac{\partial^2 y}{\partial y^2} + \frac{\partial^2 z}{\partial z^2} = f(x, y, z)$
- and so on.

20.3 Uses of FFT

Most applications of FFTs require multiplication by both F and F^{-1} . It turns out that multiplying by F and its inverse are essentially the same, because of the fact that F^{-1} is the complex conjugate of F divided by n .

For solving the Poisson equation and various other applications, we use variations on the FFT:

- The "sin" transform – the imaginary part of F
- The "cos" transform – the real part of F

Algorithms using FFT are similar, so we will focus on the forward FFT.

20.3.1 Using the 1-d FFT for filtering

Given measurements of a signal, such as $\sin 7t + .5(\sin 5t)$ at 128 points, where the noise on the signal is a random number bounded by .75, we can filter out the noise by computing the FFT of the signal, zeroing out the coefficients of the new function where the coefficient $\leq .25$, and then inverting the FFT operation. This "kills the noise," restoring the original function by zeroing out what is believed to be high-frequency noise.

20.3.2 Using the 2-d FFT for image compression

Given an image, for example, that is a 200x320 matrix of values, one can compress the image by keeping only the largest 2.5% of the coefficients of the FFT of the matrix. This allows us to discard 97.5% of the data of the image, even though the content of the image is still discernable. An example of a photo of a clown as input was given, and while some color information and clarity was visibly lost, the resulting "compressed" image was still clearly identifiable as a clown. A concept similar to this form of compression is used for jpeg images.

20.3.3 Long Multiplication

Long multiplication, as performed by grade-school students, is essentially a linear convolution of the products of each of the digits of one term with the other term. Specifically, it can be represented:

$$\text{For numbers } A, B, \text{ (and their product, } C), C_k = \sum_{j=0}^{2n-1} a_j b_{2n-j}$$

Interestingly, this is equivalent to a Fourier Transform:

$$C_k = \sum_{j=0}^{2n-1} a_j b_{2n-j} = FFT^{-1}(FFT(A) \cdot FFT(B))$$

Consequently, multiplication can be performed more efficiently than the traditional method. Specifically, their runtimes are:

- Grade School Method: $T(n) = 2n^2$
- FFT Method: $T(n) = 15n \log n$

20.4 Computing FFT

20.4.1 Recursive algorithm

FFTs can be evaluated using the divide-and-conquer recursive strategy.

- Divide the vector of coefficients into the even-index terms and the odd-index terms
- V becomes $V_{even}(x^2) + xV_{odd}(x^2)$
- V has degree $m - 1$, so V_{even} and V_{odd} are polynomials of degree $\frac{m}{2} - 1$
- So, FFT on m points is reduced to 2 FFT computations on $\frac{m}{2}$ points.
- Divide and Conquer!

20.4.2 Iterative algorithm

- Calculates FFTs in bit-reversed order:
 - 0b0000 = 0
 - 0b1000 = 8
 - 0b0100 = 4
 - 0b1100 = 12
 - 0b0010 = 2
 - ...
 - (Notice that if the bits were reversed, we would simply be counting using 4-bit binary numbers)
- Algorithm overwrites $v[i]$ with $(F \cdot v)[bitreverse(i)]$
- In many cases, though, the order doesn't matter, since as long as the inverse FFT is performed in the same order as the original FFT, the results will be equivalent. There are, however, some applications for which the order of these evaluations does matter.

20.4.3 Parallel Complexity

- M =vector size, p =number of processors
- F =time per flop = 1
- α =startup for message (in F units)
- β =time per word in a message (in F units)
- $Time(blockFFT) = Time(cyclicFFT) = \frac{2m \log m}{p} + \log(p)\alpha + \frac{m \log p}{p\beta}$

20.4.3.1 FFT with "Transpose"

If we start with a cyclic layout for the first $\log p$ steps, there need be no communication across processes. For the last $\log(\frac{m}{p})$ steps, we transpose the vector (Note: for simplicity, assume $\log(\frac{m}{p}) = \log p$. If this does not hold, more phases and layouts will be necessary). All of the communication needed for parallelization comes from the transposition of the matrix.

Essentially, the algorithm is as follows:

1. Make a matrix of x-values
2. perform a 1-d FFT using any method across each row
3. Transpose and again compute FFT across each row
4. Unzip the matrix back out into X

This works particularly well in a cache-based/hierarchical-memory system, since each row can be on a cache line. This also works well on multi-processor systems because rows can be done completely independently. For each process, the algorithm looks like this:

1. Perform Local FFTs
2. Transpose
3. Perform Local FFTs

The communication step is called a Transpose because it is analogous to transposing an array, and the same type of communication must occur if a matrix were divided across processes by rows. Furthermore, this algorithm is provably close to optimal. The LogP paper has details on why this is the case.

20.4.3.2 Higher Dimensional FFTs

FFTs on 2 or 3 dimensions are defined as 1-d FFTs on vectors in all dimensions. For example, a 2-d FFT performs 1-d FFTs on all rows and columns. There are 3 obvious possibilities for computing the 2-d FFT:

1. 2-d blocked layout for matrix, using 1-d algorithms for each row and column
2. Block row layout for the matrix, using 1-d FFTs on rows, followed by a transpose, then more serial 1-d FFTs
3. Block row layout used for matrix, using serial 1-d FFTs on rows, followed by parallel 1-d FFTs on columns

For 3-d FFTs, options for parallelization are similar.

20.5 Fastest Fourier Transform in the West (FFTW)

The Fastest Fourier Transform in the West (FFTW <http://www.fftw.org>) is an optimized FFT library. It produces an FFT implementation optimized for:

- Your version of FFT (complex, real, etc.)
- Your value of n (arbitrary, possibly prime, etc.)
- Your architecture

The implementation produced by FFTW is close to optimal for serial, but can be improved for parallel processing. The FFTW project is similar in spirit to PHIPAC, ATLAS, and Sparsity. In 1999, it won the Wilkinson Prize for Numerical Software. It has come to be widely used for serial FFTs. In version 2, it had Parallel FFTs, but FFTW no longer supports this scenario, because layout constraints from users, applications, and network differences are difficult to support. When FFTW was released, it outperformed vendor-packaged and optimized FFT calculations from Intel, AMD, etc., which was an impressive feat. Today, vendor-packaged implementations have improved, and this improvement owes a lot to FFTW's appearance. The general pattern for FFTW programs is as follows:

1. Plan the FFT (determine size, method, etc.)
2. Execute
3. Repeat execution (Note: usually, many transforms of the same size are required, so the planning step need only be done a minimal number of times)
4. Destroy plan

20.6 Bisection Bandwidth

FFT requires one (or more) transpose operations. Every processor must send $\frac{1}{P}$ of its data to each other one. As a result, bisection bandwidth (the bandwidth across the narrowest part of the network, which is important in global transpose operations, all-to-all communication, etc.) becomes a bottleneck for parallel implementations of FFT. Unfortunately, "full bisection bandwidth" is expensive. The fraction of the machine cost in the network is increasing. Fat-tree and full crossbar topologies may help to ameliorate this problem somewhat. This problem is difficult to solve, however, because communication overhead on various parallel computers varies widely (send overhead vs. receive overhead vs. send/receive overhead, etc.), making "one-size-fits-all" solutions less likely.

20.6.1 GASNet

GASNet offers put/get communication, which can be used to reduce the communication overhead. It is one-sided, meaning no remote CPU involvement is required in the API (which represents a key contrast with MPI). Instead, the message contains the remote address, there is no requirement for a matching "receive" on the target process, and no implicit ordering is required. GASNet is used in language runtimes, such as UPC. GASNet allows both Fine-grained and bulk transfers, as well as split-phase communication. Empirical data indicates that it outperforms MPI by an order of magnitude at the half-way point, but outperforms it (though not by as much) under every circumstance. It's worth noting that GASNet began as a 267-style research project.

20.6.2 Communication Strategies for 3-d FFT

Three approaches are proposed for 3-d FFTs:

1. Chunk
 - Wait for 2-d FFT to finish
 - Minimizes the number of messages
2. Slab
 - Wait for a chunk of rows destined for 1 process to finish
 - Overlap communication with computation
3. Pencil
 - Send each row as it completes
 - Maximize overlap of communication with computation
 - Match the natural layout

In NAS benchmarks, the non-blocking transfer scheme was most performant.

20.7 Takeaway

Use FFTW rather than implementing your own FFTs. Unfortunately, however, they have backed off support for parallel processing, so use FFTW with FFT in each process and use some communication strategy to accomplish the overall FFT.