# Lecture plan

- Last time: Homework, Sparse Linear Algebra
- This time: FFT (and more)

# Preliminaries: Laplace/Fourier Transform

- Laplace Transform $F(s) = \int_0^\infty f(t)e^{-st}dt$
  - $s = j\omega$, $\omega$ *continuous*
  - $t$ is also *continuous*
- Fourier Transform $F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt$

# Continuous vs. Discrete

- But computers are **discrete!**
  - Both floating point and ints are fixed resolution
  - Time must be discretized (memory has discrete addresses)
- Solutions:
  - *Quantize* continuous inputs $\rightarrow$ discrete (integer) values
  - *Samping* continuous time $\rightarrow$ discrete time

# DFT definitions

- Define $j = \sqrt{-1}$ and index matrices and vectors from 0.
- Given some number $N$, let
  $W_N = e^{-j2\pi/N} = \cos(2\pi/N) + j * \sin(2\pi/N)$
- $W$ is a complex number with whose $N$-th power $W_N^N = 1$ and is therefore called an $N$-th *root of unity*
- Used in the FFT, these powers of $W$ are also called *twiddle factors*

# Motivation for the FFT

- Signal processing
- Image processing
- Solving Poisson's Equation nearly optimally
  - $O(N \log N)$ arithmetic operations, N $=$ number of unknowns
  - Competitive with multigrid
- Fast multiplication of large integers: Schonhage-Strassen: $O(b \log b \log \log b)$ where $b =$ number of bits

# Poisson's equation arises in many models

In general, $\sum_1^N \frac{\partial^2 \phi}{\partial x_i^2} = f$

- 3D: $\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 + \partial^2 u/\partial z^2 = f(x, y, z)$
- 2D: $\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = f(x, y)$
- 1D: $d^2 u/dx^2 = f(x)$

Applications:

- Electrostatic or Gravitational Potential: Potential(position)
- Heat flow: Temperature(position, time)
- Diffusion: Concentration(position, time)
- Fluid flow: Velocity,Pressure,Density(position,time)
- Elasticity: Stress,Strain(position,time)
- Variations of Poisson have variable coefficients

# 2D Poisson with FFT

- $L_1 = F \cdot D \cdot F^T$ is eigenvalue/eigenvector decomposition, where
  - $F$ is very similar to FFT (imaginary part): $F(j,k) = (2/(n+1))^{1/2} \cdot \sin(jkp/(n+1))$
  - $D$ = diagonal matrix of eigenvalues: $D(j,j) = 2(1 - \cos(jp/(n+1)))$
- 2D Poisson same as solving $L_1 \cdot X + X \cdot L_1 = B$ where $X$ square matrix of unknowns at each grid point, $B$ square too
- Substitute $L1 = F \cdot D \cdot F^T$ into 2D Poisson to get algorithm:
  1. Perform 2D "FFT" on $B$ to get $B' = F^T \cdot B \cdot F$, or $B = F \cdot B' \cdot F^T$
  2. Solve $DX' + X'D = B'$ for $X' : X'(j,k) = B'(j,k)/(D(j,j) + D(k,k))$
  3. Perform inverse 2D "FFT" on $X' = F^T \cdot X \cdot F$ to get $X = F \cdot X' \cdot F^T$
- Cost = 2 2D-FFTs plus $n^2$ adds, divisions = $O(n^2 logn)$
- 3D Poisson analogous

# Related Transforms

- Most applications require multiplication by both $F$ and $F^{-1}$
- Multiplying by $F$ and $F^{-1}$ are essentially the same.
- $F^{-1} = F^*/m$, * is the complex conjugate
- For solving the Poisson equation and various other applications, we use variations on the FFT
  - The sin transform – imaginary part of F
  - The cos transform – real part of F
- Algorithms are similar, so we will focus on F

# Serial Algorithm for the FFT

Compute the $FFT(F * v)$ of an N-element vector $v$

$$
\begin{aligned}
(F * v)[j] &= \sum_{k=0}^{N-1} F(j,k) * v(k) \\
&= \sum_{k=0}^{N-1} W_N^{j*k} * v(k) \\
&= \sum_{k=0}^{N-1} (W_N^j)^k * v(k) \\
&= V(W_N^j)
\end{aligned}
$$

where V is defined as the polynomial $V(x) = \sum_{k=0}^{N-1} x^k * v(k)$
So, the FFT is the *same* as evaluating a polynomial $V(x)$ with degree $N-1$ at $N$ different points

# Divide and Conquer FFT

V can be evaluated using divide-and-conquer:

$$
\begin{aligned}
V(x) &= \sum_{k=0}^{N-1} x^k * v(k) && (1)\\
&= v[0] + x2 * v[2] + x4 * v[4] + \ldots + && (2)\\
& \quad x * (v[1] + x2 * v[3] + x4 * v[5] + \ldots) && (3)\\
&= V_{even}(x2) + x * V_{odd}(x2) && (4)\\
& && (5)
\end{aligned}
$$

- $V$ has degree $N-1$, so $V_{even}$ and $V_{odd}$ are polynomials of degree $N/2-1$
- We evaluate these at $N$ points: $(W_N^l)^2$ for $0 \leq l \leq m-1$
- But this is really just $N/2$ different points, since $(W^(l + N/2))^2 = (W^l * W^{N/2})^2 = W^{2l} * W^N = (W^l)^2$
- So FFT on $N$ points reduced to 2 FFTs on $N/2$ points

*Divide and conquer!*

# Divide and Conquer FFT

FFT(v, W, N) ... assume N is a power of 2
    if N = 1 return v[0]
    else
        veven = FFT(v[0:2:N-2], $W_N^2, N/2$)
        vodd = FFT(v[1:2:N-1], $W_N^2, N/2$)
        Wvec = [$W_N^0, W_N^1, \ldots W_N^{(N/2-1)}$ ]
        return [veven + (Wvec .* vodd ),
        veven - (Wvec .* vodd ) ]

Cost: $T(N) = 2T(N/2) + O(N) = O(N \log N)$ operations

Twiddle factors are *precomputed*

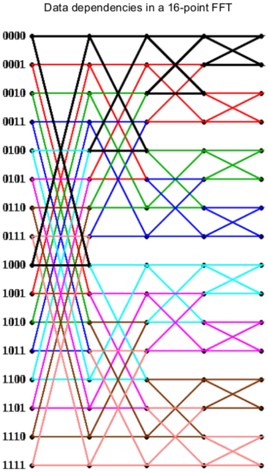
# Examining the call tree

- Divides by even and odd: low order bit
  1. level 0: xxx0, xxx1
  2. level 1: xx00, xx10 — xx01, xx11
  3. etc... etc...
- This is *bit reversed*!
- An iterative algorithm that uses loops rather than recursion, does each level in the tree starting at the bottom
- Algorithm overwrites v[i] by (F*v)[bitreverse(i)]
- Practical algorithms combine recursion (for memory hierarchy) and iteration

# Bit reversal

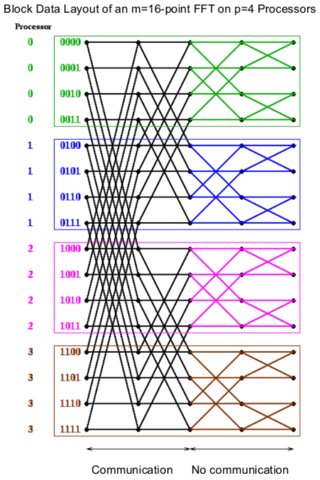
- The outputs of the FFT stored in *bit reversed* order
- May need to sort them (which is like transposes at various levels)
- Many FFT uses will do a reverse FFT as well, and therefore not require a bit reversal at all
- DSP processors offer bit reversed addressing

# FFT by stage



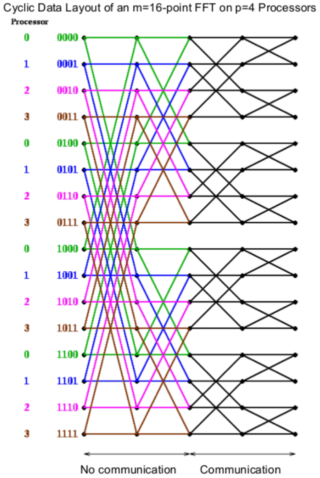
Data dependencies in a 16-point FFT
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

# FFT by stage: 4 processor mapping




- Using a block layout ($N/p$ contiguous words per processor)
- No communication in last $\log N/p$ steps
- Significant communication in first $\log p$ steps

# FFT by stage: cyclic mapping



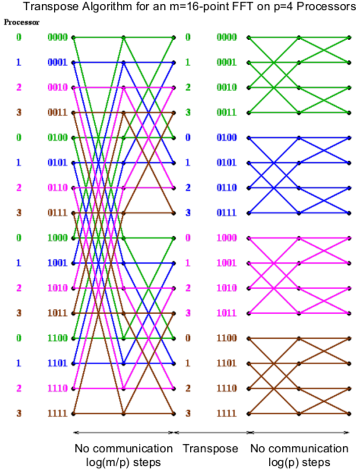

- Cyclic layout (consecutive words map to consecutive processors)
- No communication in first $\log(N/p)$ steps
- Communication in last $\log(p)$ steps

# Parallel Complexity

- $N$ = vector size, $p$ = number of processors
- $f$ = time per flop = 1
- $a$ = latency for message
- $b$ = time per word in a message

Time(blockFFT) = Time(cyclicFFT) =
$2 * N * \log(N)/p$ ... perfectly parallel flops
$+ \log(p) * a$ ... 1 message/stage, log p stages
$+ N * \log(p)/p * b$ ... N/p words/message

# FFT by stage: transpose mapping




- Start with a cyclic layout for first $\log(N/p)$ steps, there is no communication
- Then transpose the vector for last $\log(p)$ steps
- All communication is in the transpose

Communication analogous to transposing an array

# Sequential Communication Complexity of the FFT

How many words need to be moved between main memory and cache of size $M$ to do the FFT of size $N$, where $N > M$?

- Theorem (Hong, Kung, 1981): words $= \Omega(N \log N / \log M)$: Proof follows from each word of data being reusable only $\log M$ times; assumes no recomputation
- Attained by transpose algorithm
  - Sequential algorithm "simulates" parallel algorithm
  - Imagine we have $P = N/M$ processors, so each processor stores and works on $O(M)$ words
  - Each local computation phase in parallel FFT replaced by similar phase working on cache resident data in sequential FFT
  - Each communication phase in parallel FFT replaced by reading/writing data from/to cache in sequential FFT
- Attained by recursive, "cache-oblivious" algorithm (FFTW)

# Higher Dimensional FFTs

- FFTs on two or more dimensions are defined as 1D FFTs on vectors in all dimensions.
- 2D FFT does 1D FFTs on all rows and then all columns
- There are 3 obvious possibilities for the 2D FFT:
    1. 2D blocked layout for matrix, using parallel 1D FFTs for each row and column
    2. Block row layout for matrix, using serial 1D FFTs on rows, followed by a transpose, then more serial 1D FFTs
    3. Block row layout for matrix, using serial 1D FFTs on rows, followed by parallel 1D FFTs on columns
- Option 2 is best, if we overlap communication and computation
- For a 3D FFT the options are similar
    - Two phases done with serial FFTs, followed by a transpose for 3rd
    - Can overlap communication with 2nd phase in practice

# Bisection Bandwidth

- FFT requires one (or more) transpose operations: Every processor sends 1/p-th of its data to each other one
- Bisection Bandwidth limits this performance
  - Bisection bandwidth is the bandwidth across the narrowest part of the network
  - Important in global transpose operations, all-to-all, etc.
- "Full bisection bandwidth" is expensive
- Fraction of machine cost in the network is increasing
  - Fat-tree and full crossbar topologies may be too expensive
  - Especially on machines with 100K and more processors
  - SMP clusters often limit bandwidth at the node level
- Goal: overlap communication and computation

# Fourier Transform Benchmark Case Study

- Performance of Exchange (All-to-all) is critical
  - Communication to computation ratio increases with faster, more optimized 1-D FFTs (used best available, from FFTW)
  - Determined by available bisection bandwidth
  - Between 30-40% of the application's total runtime
- Assumptions
  - 1D partition of 3D grid
  - At most N processors for $N^3$ grid
  - HPC Challenge benchmark has large 1D FFT (can be viewed as 3D or more with proper roots of unity)

# 3D FFTs

- $NX \times NY \times NZ$ elements spread across P processors
- Each processor gets $NZ/P$ "planes" of $NX \times NY$ elements per plane
- 3 step process:
  1. FFTs on the columns (all elements local)
  2. FFTs on the rows (all elements local)
  3. FFTs in the Z-dimension (requires communication)

# Parallel Processing

- Each processor has to scatter input domain to other processors
  - Every processor divides its portion of the domain into P pieces
  - Send each of the P pieces to a different processor
- Three different ways to break up the messages:
  1. Chunk (i.e. single packed "All-to-all" in MPI parlance) (3D) = all rows with same destination
  2. Slabs (2D) = all rows in a single plane with same destination
  3. Pencils (1D) = 1 row
- Going from Chunks to Slabs to Pencils leads to
  - An order of magnitude increase in the number of messages
  - An order of magnitude decrease in the size of each message
- Why do this? Slabs and Pencils allow *overlapping* communication and computation

# Decomposing NAS FT Exchange into Smaller Messages

Three approaches:

1. Chunk: Wait for 2D FFTs to finish
2. Slab: Wait for chunk of rows destined for 1 proc to finish
3. Pencil: Send each row as it completes

Example Message Size Breakdown for Class D (2048 x 1024 x 1024) with 256 processors

| | |
|---|---|
| Chunk | 512 Kbytes |
| Slabs | 65 Kbytes |
| Pencils | 16 Kbytes |

# FFTW: Fastest Fourier Transform in the West

`www.fftw.org`

- Produces FFT implementation optimized for
  - Your version of FFT (complex, real,...)
  - Your value of N (arbitrary, possibly prime)
  - Your architecture
  - Very good sequential performance
- Won 1999 Wilkinson Prize for Numerical Software
- Widely used
  - Latest version 3.3.10
  - Includes threads, OpenMP, MPI versions, new architecture support
  - Layout constraints from users/apps + network differences are hard to support

# More FFTW

- C library for real and complex FFTs (arbitrary size/dimensionality)
- Computational kernels (80% of code) automatically generated
- Self-optimizes for your hardware (picks best composition of steps) = portability + performance
- Exploits CPU-specific SIMD instructions (rewriting the code)
- FFTW implements many FFT algorithms: A planner picks the best composition by measuring the speed of different combination
- Free! Gnu!

# FFTW selling points

- Speed. (Supports SSE/SSE2/Altivec)
- Both one-dimensional and multi-dimensional transforms.
- Arbitrary-size transforms. (Sizes with small prime factors are best, but FFTW uses O(N log N) algorithms even for prime sizes.)
- Fast transforms of purely real input or output data.
- Transforms of real even/odd data: the discrete cosine transform (DCT) and the discrete sine transform (DST).
- Efficient handling of multiple, strided transforms.
- Parallel transforms: parallelized code for platforms with SMP machines with some flavor of threads (e.g. POSIX) or OpenMP. Also MPI.
- Portable to any platform with a C compiler.
- Both C and Fortran interfaces.