# Numerical Analysis, High Performance Computing, Big Data, Quantum Computing Review

*Ron Wu*

Last update: 8/30/16

## Contents

The Past: Numerical Analysis ........ 3

1. Ordinary Differential Equation ........ 3
   - Initial Value Problem, Euler Method, Runge-Kutta, Adams-Bashforth… ........ 3
   - Stability, Error Analysis, Convergence ........ 6
   - Boundary Value Problem ........ 7
2. Finite Difference Scheme for PDE ........ 7
   - Heat Equation ........ 7
   - Neumann Analysis ........ 8
   - Transport Equation ........ 8
3. Solve Ax=b ........ 11
   - 2D Heat Equation ........ 11
   - Gaussian elimination, Sparse LU decomposition ........ 13
   - Other Iteration Methods for Special Matrix ........ 15
4. Spectral Method for PDE ........ 16
   - Fast Fourier Transform (FFT) ........ 16
   - Fast Chebyshev Transform ........ 21
   - Fast Multipole Method (FMM) ........ 24
   - Other Transforms ........ 25

The Present: High Performance Computing ........ 26

5. PRAM Model ........ 26
   - Brent’s Theorem ........ 26
   - Parallel Loop ........ 27

- Parallel Sort ..... 28
- Parallel Linked List ..... 31
- Parallel Tree ..... 33
- Parallel Graph ..... 34

6. Distributed Memory Model ..... 36

- Message-Passing Model ..... 36
- Distributed Operations ..... 37
- Distributed Network Topology ..... 40
- Distributed Matrix Multiplication ..... 41
- Distributed Sort ..... 44
- Distributed Graph Operations ..... 47

7. Memory/CPU Optimization ..... 51

- Two-Level Memory Hierarchy ..... 52
- Two-Level Matrix Multiplication ..... 52
- Two-Level Memory Sorting ..... 54
- Two-Level Memory Searching ..... 57
- Cache and Paging ..... 58
- Cache Oblivious Algorithms ..... 60

The Future: Quantum Computing ..... 61

8. Quantum Information & Entanglement ..... 61

- Mix States ..... 61
- Entropy and Information ..... 63
- Entanglement ..... 63

9. From Laser to Quantum Computer ..... 64

- Coherent Mixture States & Lasers ..... 65
- Quantum Transistor and Gates ..... 66

10. Quantum Algorithms ..... 67

- Deutsch’s Problem ..... 67
- Grover Algorithm ..... 68
- Quantum Fourier Transform ..... 69
- Shor Algorithm (factorization) ..... 70

Reference......70
11. Books......70
12. Courses......70
13. On-line Course......71
14. Computing......71

A list of great numerical algorithms and history
http://people.maths.ox.ac.uk/trefethen/inventorstalk.pdf

The HPC notes are based on an on-line course by Rich Vuduc, *High Performance Computing*, Udacity, George Tech.

## The Past: Numerical Analysis

### 1. Ordinary Differential Equation

- Initial Value Problem, Euler Method, Runge-Kutta, Adams-Bashforth…

$$y'(t) = f\big(t, y(t)\big), \qquad y(t_0) = y_0$$

Forward Euler (Explicit) ~ 1st order ~ local truncation error $O(h^2)$

$$y_{n+1} = y_n + hf(t_n, y_n)$$

Proof:

$$y_{n+1} = y(t_n + h) = y_n + hy'(t_n) + O(h^2)$$

Thus 1st order schema means its local error $O(h^2)$. Local error is the error directly inherited from its previous $y_n$. Since every step has some error, what is the total error contributed up to this point? We define global truncation error

$$g_n = |y_{true}(t_n) - y_n|$$

Claim $g_n \sim O(h)$.

Proof: Assume $f$ is Lipschitz in $y$ with Lipschitz constant $L$, and $|y''|$ is bounded by $M$.

$$y_{true}(t_{n+1}) = y_{true}(t_n) + hf\big(t_n, y_{true}(t_n)\big) + \underbrace{\frac{h^2}{2} y''(\tau)\bigg|_{\tau\in[t_n,t_{n+1}]}}_{|\quad|\le Mh^2}$$

$$hf\big(t_n, y_{true}(t_n)\big) \le \underbrace{hf(t_n, y_n)}_{y_{n+1}-y_n} + h\,\underbrace{[y_{true}(t_n) - y_n]}_{\pm g_n}\,L$$

Combining the two above

$$y_{true}(t_{n+1}) - y_{n+1} \le y_{true}(t_n) - y_n + hg_nL + Mh^2$$

That is

$$g_{n+1} \le g_n(1 + hL) + Mh^2$$

Since no error at initial point, $g_1$ is equal to the local truncation error

$$g_1 = Mh^2$$

then

$$g_n \le Mh^2\,\underbrace{[1 + (1 + hL) + \cdots + (1 + hL)^{n-1}]}_{[(1+hL)^n-1]/hL} = \frac{Mh}{L}\left[(1 + hL)^{\frac{t_n}{h}} - 1\right] \to \frac{Mh}{L}[e^{Lt_n} - 1]$$

because $n = t_n/h$. We see indeed the global error grows like $O(h)$ and it grows as $t$ increases.

QED

There is another error called, Round off Error. It is different from truncation error. It is the limit of the machine representation. See IEEE standard, e.g. $10^{-16}$ for double precision. When $h \to 0$, round off error actually → infinity.

For details read short book by Michael Overton, NYU, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM

Backward Euler (Implicit) because $f(t_{n+1}, y_{n+1})$ is not known

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$$

For special $f$, one can use simple algebra to solve for $y_{n+1}$, then do iteration. But in general, we are not luckily. Then we have to update

$$y_{n+1}^{(k+1)} = y_n^{(k+1)} + hf(t_{n+1}, y_{n+1}^{(k)})$$

called fixed point iteration. Its local and global errors can be proved as before.

Trapezoidal rule/Crank-Nicolson `ode23t`
It is 2nd order

$$y_{n+1} = y_n + h\frac{f(t_{n+1}, y_{n+1}) + f(t_n, y_n)}{2}$$

2nd order Runge–Kutta `ode23`

$$y_{n+1} = y_n + hf\left(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hf(t_n, y_n)\right)$$

4th order Runge–Kutta ~ local error $O(h^5)$, global error $O(h^4)$ `ode45`

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Where

$$\begin{aligned} k_1 &= f(t_n, y_n), \\ k_2 &= f(t_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}k_1), \\ k_3 &= f(t_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}k_2), \\ k_4 &= f(t_n + h, y_n + hk_3). \end{aligned}$$

default error $10^{-6}$, change setting `odeset`
http://www.mathworks.com/help/matlab/ref/ode45.html#input_argument_options

Adams-bashforth (2-step linear fit)
Notice

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} y'(t)\,dt\,.$$

Using Newton's idea, we can use interpolating polynomial $P(t)$ as an approximation of $y'$, then to predict future $y(t_{n+1})$

$$y(t_{n+1}) \approx y(t_n) + \underbrace{\frac{3}{2}hf(t_n, y_n) - \frac{1}{2}hf(t_{n-1}, y_{n-1})}_{:=P_{n+1}}$$

Two-step Adams-Moulton (implicit scheme) uses Adams-bashforth result as an intermediate step, then fed into the implicit Euler, called the corrector step. This method is very good for stiffness (involving high derivative, i.e. small $h$ results large oscillation—unstable, so implicit scheme becomes a better choice, which allows to take larger time steps than any other schemes, we will prove this later) `ode113`

First compute $P_{n+1}$ as above, then backward Euler

$$y_{n+1} = y_n + hf(t_{n+1}, y_n + P_{n+1})$$

Or half backward and half forward

$$y_{n+1} = y_n + h\frac{f(t_{n+1}, y_n + P_{n+1}) + f(t_n, y_n)}{2}$$

Gear method `ode15s` for stiffness rising from physics

http://www.mathworks.com/help/matlab/math/choose-an-ode-solver.html#bu3n5rf-1

| **Solver** | **Problem Type** | **Accuracy** | **When to Use** |
|---|---|---|---|
| `ode45` | Nonstiff | Medium | Most of the time. `ode45` should be the first solver to try. |
| `ode23` | | Low | `ode23` can be more efficient than `ode45` at problems with crude tolerances, or in the presence of moderate stiffness. |
| `ode113` | | Low to High | `ode113` can be more efficient than `ode45` at problems with stringent error tolerances, or when the ODE function is expensive to evaluate. |
| `ode15s` | Stiff | Low to Medium | Try `ode15s` when `ode45` fails or is inefficient and you suspect that the problem is stiff. Also use `ode15s` when solving differential algebraic equations (DAEs). |
| `ode23s` | | Low | `ode23s` can be more efficient than `ode15s` at problems with crude error tolerances. It can solve some stiff problems for which `ode15s` is not effective.<br>`ode23s` computes the Jacobian in each step, so it is beneficial to provide the Jacobian via `odeset` to maximize efficiency and accuracy.<br>If there is a mass matrix, it must be constant. |
| `ode23t` | | Low | Use `ode23t` if the problem is only moderately stiff and you need a solution without numerical damping.<br>`ode23t` can solve differential algebraic equations (DAEs). |
| `ode23tb` | | Low | Like `ode23s`, the `ode23tb` solver might be more efficient than `ode15s` at problems with crude error tolerances. |
| `ode15i` | Fully implicit | Low | Use `ode15i` for fully implicit problems *f(t,y,y') = 0* and for differential algebraic equations (DAEs) of index 1. |

- Stability, Error Analysis, Convergence

Just like to test stability of linear system $y_n = Ay_{n-1}$, ( $solution$: stable if all eigenvalues of $A <$ 1), here to test the stability of for(back)word Euler, we try linear function

$$f = y' = ay$$

(notice this satisfies the Lipschitz condition we discussed before)

Let $a$ be complex, then forward Euler

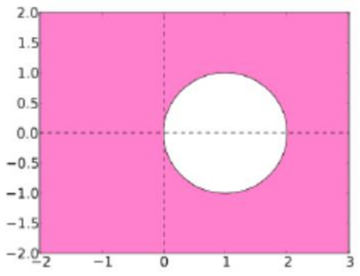
$$y_{n+1} = (1 + ah)y_n = (1 + ah)^{n+1}y_0$$

Backward Euler

$$y_{n+1} = \frac{1}{1 - ah}y_n = \frac{1}{(1 - ah)^{n+1}}y_0$$

Hence the solution will not grow to $\infty$ for small $y_0$, i.e

$$\text{stable, if } |1 \pm ah| \begin{cases} < 1 & forward \\ > 1 & backward \end{cases}$$



Back(for)ward Euler, (un)stable in pink region

Showing the stability region for backward Euler is very big, so implicit scheme allows large $h$

- Boundary Value Problem

Dirichlet, or even Robin = Dirichlet + Neumann, e.g. SHO of two ends

$$y''(x) + y(x) = 0, y(0) = 0, y'\left(\frac{\pi}{2}\right) = 1$$

Apply shooting method

Use $y_0$ and try different $y'_0$ till hit $y_T$
Infinite Domain: boundary given at infinity
Turn to Neumann at large $T$, Turn into system of 1st order
For linear ode, use finite difference then get to $Ax = b$

bvp4c http://mathworks.com/help/matlab/ref/bvp4c.html#zmw57dd0e79452

## 2. Finite Difference Scheme for PDE

- Heat Equation

Heat equation is very important because Black-Scholes is a heat equation, see http://www.math.nyu.edu/faculty/kohn/pde.finance/2015/section2.pdf

$$u_t = u_{xx}$$

- Neumann Analysis

Find the bound of Courant–Friedrichs–Lewy condition (CFL) so that the solution is stable in time.

$$\lambda = \frac{\Delta t}{\Delta x^2} \leq \text{CFL}$$

Leapfrog (2, 2)

$$u_n^{m+1} = u_n^{m-1} + 2\lambda(u_{n+1}^m - 2u_n^m + u_{n-1}^m)$$

resulting $|g| > 1$ unstable for all $\lambda$. $g$ is given later in the proof.

Forward Euler

$$u_n^{m+1} = u_n^m + \lambda(u_{n+1}^m - 2u_n^m + u_{n-1}^m)$$

resulting $|g| \leq 1$ stable for $\lambda \leq \frac{1}{2}$.

Backward Euler

$$u_n^{m+1} = u_n^m + \lambda(u_{n+1}^{m+1} - 2u_n^{m+1} + u_{n-1}^{m+1})$$

- Transport Equation (One-way Wave Equation)

It is the simplest pde.

$$u_t = u_x$$

$$\text{CFL}\ \ \lambda = \frac{\Delta t}{\Delta x}$$

Forward Euler

$$u_n^{m+1} = u_n^m + \frac{1}{2}\lambda(u_{n+1}^m - u_{n-1}^m)$$

*m* temporal step, *n* spatial step, unstable for all λ.

Leapfrog (2, 2)

$$u_n^{m+1} = u_n^{m-1} + \lambda(u_{n+1}^m - u_{n-1}^m)$$

stable for λ ≤ 1.

Leapfrog (2, 4) ~ $O(\Delta x^5)$

$$u_n^{m+1} = u_n^{m-1} + \lambda\left[\frac{4}{3}(u_{n+1}^m - u_{n-1}^m) - \frac{1}{6}(u_{n+2}^m - u_{n-2}^m)\right]$$

stable for $\lambda \leq 0.7$. 100 times faster than leapfrog (2, 2)

Lax-Wendroff

$$u_n^{m+1} = u_n^m + \frac{1}{2}\lambda(u_{n+1}^m - u_{n-1}^m) + \frac{1}{2}\lambda^2(u_{n+1}^m - 2u_n^m + u_{n-1}^m)$$

add diffusion to Euler forward, making it stable.

Implicit Schemes

$$u_n^{m+1} = u_n^m + \frac{1}{2}\lambda(u_{n+1}^{m+1} - u_{n-1}^{m+1})$$
$$Au^{m+1} = u^m$$

where matrix $A = \begin{pmatrix} 2 & -\lambda & \cdots & \\ \lambda & 2 & & \\ & \vdots & \ddots & \vdots \\ & & \cdots & \end{pmatrix}$. stable for all $\lambda$.

MacCormick Predictor Corrector

$$u_n^p = u_n^m + \frac{\lambda}{2}(u_{n+1}^m - u_{n-1}^m)$$
$$u_n^{m+1} = \frac{1}{2}[u_n^m + u_n^p + \lambda(u_{n+1}^p - u_{n-1}^p)]$$

Proof of Stability
The Neumann Analysis we did before won't work for non-linear, and it will provide necessary not sufficient stable condition.

Assume solution of the form

$$y_n^m = g^m e^{in\xi\Delta x}$$

This is very typical solution. Temporal grows like Gaussian and spatial functions are sinusoid so that boundaries can be easily satisfied.

Proof:
Recall general solution to heat equation. First Fourier transform of the heat equation, converting it to an algebraic equation

$$u_t = u_{xx}$$

$$\frac{1}{\sqrt{2\pi}}\left(\underbrace{\int_{-\infty}^{\infty} u_t(x,t)e^{-ixp}dx}_{=\frac{\partial \hat{u}(p,t)}{\partial t}} = \int_{-\infty}^{\infty} u_{xx}(x,t)e^{-ixp}dx = \underbrace{u_x e^{-ixp}\Big|_{-\infty}^{\infty}}_{0} + ip\int_{-\infty}^{\infty} u_x(x,t)e^{-ixp}dx\right.$$

$$\left. = ip\,\underbrace{ue^{-ixp}\Big|_{-\infty}^{\infty}}_{0} - p^2\underbrace{\int_{-\infty}^{\infty} u(x,t)e^{-ixp}dx}_{=\hat{u}(p,t)}\right)$$

So

$$\hat{u}(p,t) = \hat{u}(p,0)e^{-p^2 t}$$

then inverse Fourier

$$u(x,t) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}\hat{u}(p,0)e^{-p^2 t}e^{ixp}dp$$

When $t = 0$,

$$u(x,t=0) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}\hat{u}(p,0)e^{ixp}dp = u(x,0)$$

Hence we verified $\hat{u}(p,0) = F[u(x,0)]$. Recall Fourier of a Gaussian is a Gaussian, and product of Fourier is convolution,

$$\begin{aligned}
u(x,t) &= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}u(x',0)e^{-ix'p}\,dx' e^{-p^2 t}e^{ixp}dp \\
&= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}u(x',0)\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}\underbrace{e^{-p^2 t}e^{-i(x'-x)p}}_{e^{-\left[p\sqrt{t}+\frac{i(x'-x)}{2\sqrt{t}}\right]^2-\frac{(x'-x)^2}{4t}}}\,dpdx' \\
&= \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}u(x',0)e^{-\frac{(x'-x)^2}{4t}}\frac{1}{\sqrt{2t}}\underbrace{\frac{1}{\sqrt{\pi}}\int_{-\infty}^{\infty}e^{-\left[p\sqrt{t}+\frac{i(x'-x)}{2\sqrt{t}}\right]^2}d(p\sqrt{t})}_{1}\,dx' \\
&= \frac{1}{\sqrt{2\pi}\sqrt{2t}}\int_{-\infty}^{\infty}u(y-x,0)e^{-\frac{1}{2}\frac{y^2}{\left(\sqrt{2t}\right)^2}}dy
\end{aligned}$$

QED

If $\lim_{m\to\infty} |g|^m < \infty$, i.e. $|g| \le 1$, solution is stable in time. Substitute this into above schemes, say, heat forward Euler, we get

$$|g| = 1 + 2\lambda(\cos \xi \Delta \mathrm{x} - 1)$$

Hence stable for $\lambda \le 1/2$.

If the same solution form $y_n^m = g^m e^{in\xi\Delta x}$ applied to transport forward Euler, we get

$$|g| = \sqrt{1 + \lambda^2 \sin^2 \xi \Delta \mathrm{x}}$$

hence, unstable for all $\lambda$.

Substitute into transport implicit scheme

$$|g| = \frac{1}{\sqrt{1 + \lambda^2 \sin^2 \xi \Delta \mathrm{x}}}$$

Stable for all $\lambda$.

We know the analytic solution to transport equation is

$$y(x,t) = f(x+t)$$

That is

$$g = e^{i\xi\Delta t} = e^{i\xi\lambda\Delta x} \approx 1 + i \sin \lambda \xi \Delta x$$

## 3. Solve Ax=b

Up to now we applied numerical algorithm in looping iterations. Contemporarily one should solve them simultaneously. Hence avoiding loops, and taking advantage of parallelism (Matlab, R's native Fortran, C for matrix algorithms). Later we will write HPC code on our own.

- ### 2D Heat Equation

Example: solve 2d heat (Laplace) equation, with periodic boundary condition

$$\Delta u = u_t$$

$$\begin{aligned} u_{i,j}^{m+1} &= u_n^m + \lambda[\underbrace{u_{i+1,j}^m - 2u_{i,j}^m + u_{i-1,j}^m}_{1D\ forwarrd\ Euler\ in\ x-axis} + u_{i,j+1}^m - 2u_{i,j}^m + u_{i,j-1}^m] \\ &= u_n^m + \lambda\left(u_{i\pm1,j}^m + u_{i,j\pm1}^m - 4u_{i,j}^m\right) \end{aligned}$$

by forward Euler. Create mesh $\Delta x = \Delta y$ , $\lambda = \Delta t / \Delta x^2 < 1/2$ , then to standardize matrix operations, we create vectors for $u$

$$\vec{u}^m = \begin{pmatrix} \begin{pmatrix} \vdots \\ u_{1,j}^m \\ \vdots \end{pmatrix} \\ \begin{pmatrix} \vdots \\ u_{2,j}^m \\ \vdots \end{pmatrix} \end{pmatrix}$$

Then forward Euler becomes

$$u^{m+1} = I + \lambda L u^m$$

where $L =$

Matrix for Discrete Poisson Problem

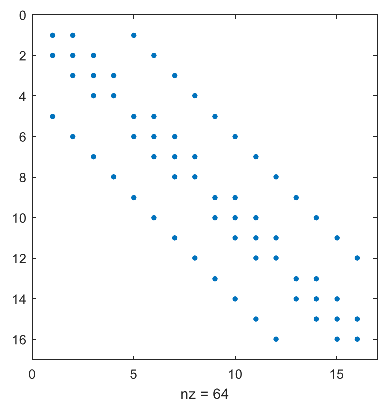
$$\left[\begin{array}{cccc:cccc:cccc:cccc}
\mathbf{4} & -1 & & & -1 & & & & & & & & & & & \\
-1 & \mathbf{4} & -1 & & & -1 & & & & & & & & & & \\
 & -1 & \mathbf{4} & -1 & & & -1 & & & & & & & & & \\
 & & -1 & \mathbf{4} & & & & -1 & & & & & & & & \\
\hdashline
-1 & & & & \mathbf{4} & -1 & & & -1 & & & & & & & \\
 & -1 & & & -1 & \mathbf{4} & -1 & & & -1 & & & & & & \\
 & & -1 & & & -1 & \mathbf{4} & -1 & & & -1 & & & & & \\
 & & & -1 & & & -1 & \mathbf{4} & & & & -1 & & & & \\
\hdashline
 & & & & -1 & & & & \mathbf{4} & -1 & & & -1 & & & \\
 & & & & & -1 & & & -1 & \mathbf{4} & -1 & & & -1 & & \\
 & & & & & & -1 & & & -1 & \mathbf{4} & -1 & & & -1 & \\
 & & & & & & & -1 & & & -1 & \mathbf{4} & & & & -1 \\
\hdashline
 & & & & & & & & -1 & & & & \mathbf{4} & -1 & & \\
 & & & & & & & & & -1 & & & -1 & \mathbf{4} & -1 & \\
 & & & & & & & & & & -1 & & & -1 & \mathbf{4} & -1 \\
 & & & & & & & & & & & -1 & & & -1 & \mathbf{4}
\end{array}\right]$$

This $L$ is a $16 \times 16$ matrix, but it only represents a spatial domain of $4 \times 4$.
This shows the need for supercomputers because serial computers cannot handle spatial size like $10^3 \times 10^3$. Luckily the matrix is very sparse.

If we used backward Euler, we would have to do inverse $L$. In sum we standardized the process of solving pde to tasks of finding eigenvalues, do matrix multiplications, and inverses.

Here is how we create the Laplacian matrix (using sparse for optimization) in Matlab

```
m = 4; n = m*m;

e1 = ones(n,1);  e2 = e1;
for j = 1:m
    e2(m*j) = 0;
end
e3(1,1)=1; e3(2:n,1)=e2(1:n-1,1);

L = spdiags([-1*e1 -1*e2 4*e1 -1*e3 -1*e1 ],[-m -1 0 1 m],n,n);
spy(L)
cond(L)
```



Later we will show MPI implementation
https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_heat2D.c

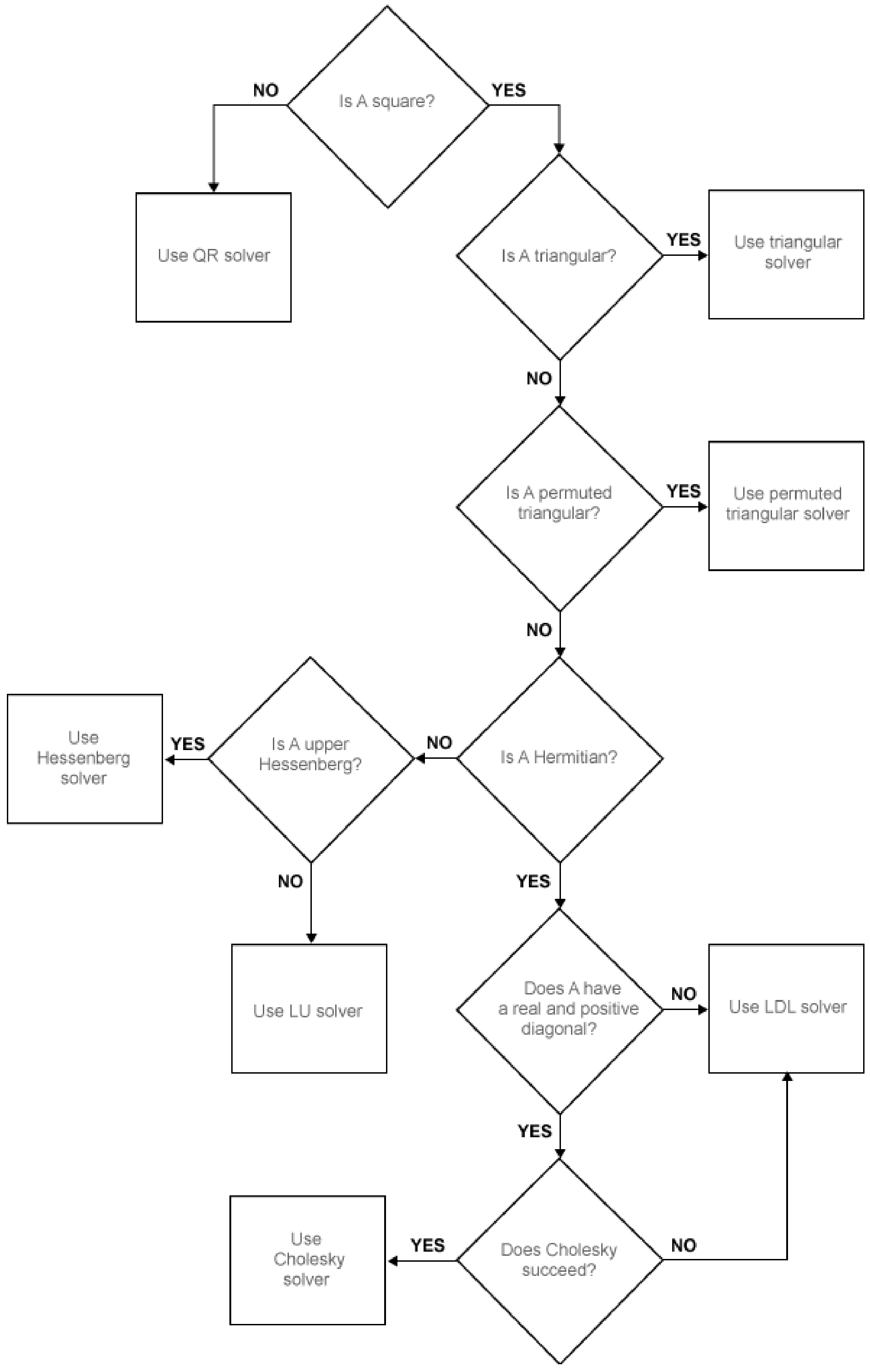
- Gaussian elimination, Sparse LU decomposition

Gaussian ~$O(N^3)$

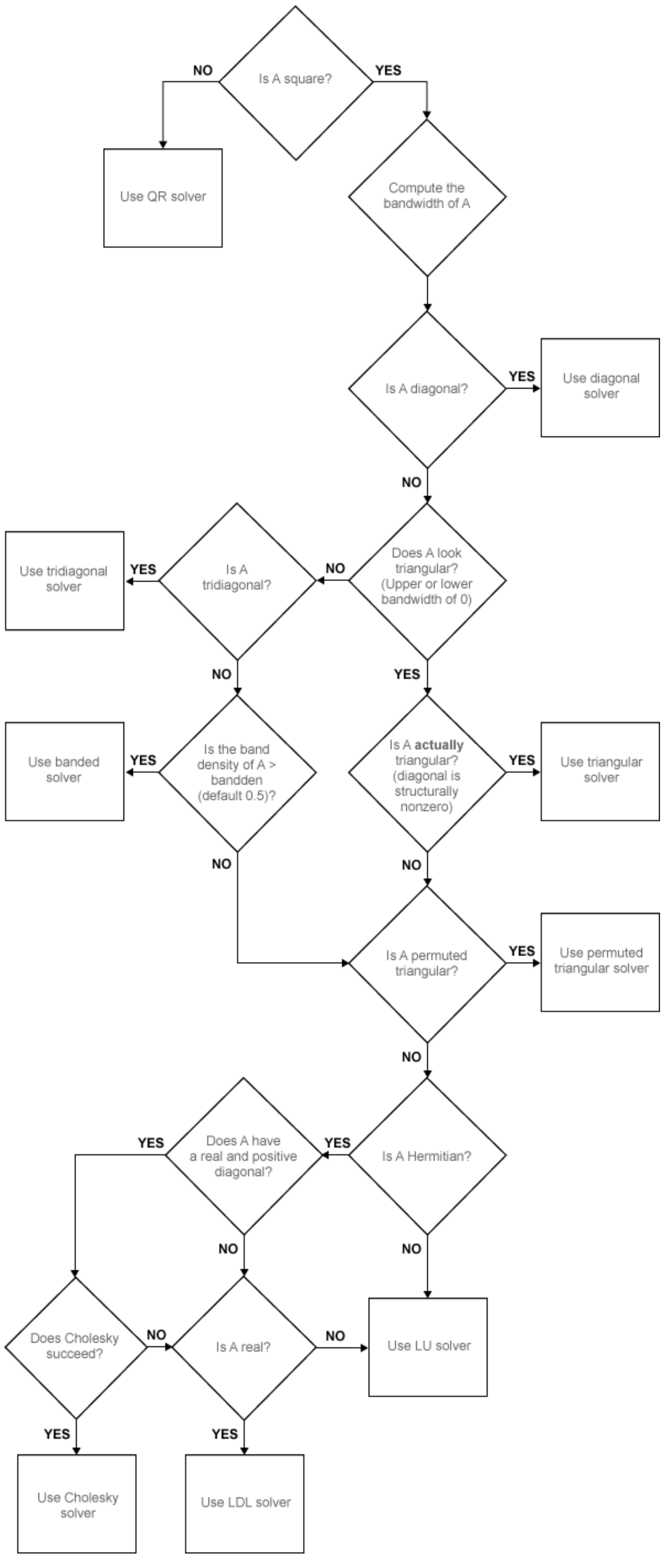
Matlab backslash

$$Ax = b => x = A\backslash b$$

It will optimize as follows. LU will be the worst case.
http://www.mathworks.com/help/matlab/ref/mldivide.html#1002049


Is A square?
NO
YES
Use QR solver
Is A triangular?
YES
Use triangular solver
NO
Is A permuted triangular?
YES
Use permuted triangular solver
NO
Is A Hermitian?
NO
Is A upper Hessenberg?
YES
Use Hessenberg solver
NO
Use LU solver
YES
Does A have a real and positive diagonal?
NO
Use LDL solver
YES
Does Cholesky succeed?
YES
Use Cholesky solver
NO

For sparse


Is A square?
NO
YES
Use QR solver
Compute the bandwidth of A
Is A diagonal?
YES
Use diagonal solver
NO
Does A look triangular? (Upper or lower bandwidth of 0)
NO
Is A tridiagonal?
YES
Use tridiagonal solver
NO
Is the band density of A > bandden (default 0.5)?
YES
Use banded solver
NO
YES
Is A **actually** triangular? (diagonal is structurally nonzero)
YES
Use triangular solver
NO
Is A permuted triangular?
YES
Use permuted triangular solver
NO
Is A Hermitian?
YES
Does A have a real and positive diagonal?
YES
NO
Does Cholesky succeed?
NO
Is A real?
NO
Use LU solver
YES
Use Cholesky solver
YES
Use LDL solver
NO

- Other Iteration Methods for Special Matrix

- Jacobi / Gauss-Seidel Iteration for strictly diagonal dominance matrix

$$|A_{i\,i}| > \sum_{j\neq i} |A_{i\,j}|$$

for all $i$. Then Jacobi Iteration converges.

It is ~ $O(NSK)$
$S$ = sparse stencil, $K$ = # iterations

- Generalized minimal residual method
http://www.mathworks.com/help/matlab/ref/gmres.html?searchHighlight=gmres#syntax

```
x = gmres(A,b)
```

- Biconjugate gradient stabilized method
http://www.mathworks.com/help/matlab/ref/bicgstab.html#syntax

```
x = bicgstab(A,b)
```

## 4. Spectral Method for PDE

- Fast Fourier Transform (FFT)

O(N log N)

Fourier Series

Recall Fourier series on a domain $[0, L]$ for periodic functions

$$f(x) = \sum_{n=-\infty}^{+\infty} c_n e^{i\frac{2\pi nx}{L}}$$

where

$$c_n = \frac{1}{L}\int_0^L f(x)\, e^{-i\frac{2\pi nx}{L}} dx$$

Recall Fourier in real form. On MatLab its Sine/cosine Transform (`dst/dct`) and their inverse (`idst/idct`),

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos\left(\frac{2\pi nx}{L}\right) + b_n \sin\left(\frac{2\pi nx}{L}\right)$$

Where

$$a_n = \frac{2}{L}\int_0^L f(x)\cos\left(\frac{2\pi nx}{L}\right)dx, \qquad b_n = \frac{2}{L}\int_0^L f(x)\sin\left(\frac{2\pi nx}{L}\right)dx$$

Their mapping is

$$c_n = \frac{a_n}{2} - i\frac{b_n}{2}, n \geq 0, \qquad c_{-n} = -c_n^*$$

Or

$$a_n = 2(c_n + c_{-n}), \qquad b_n = 2(c_n - c_{-n})i$$

so that $c_n$ is complex but $f(x)$ will be real. So to draw the Fourier of $f(x)$, just take the real/imaginary of $c_n$. Since the real sine/cosine Fourier is equivalent to the complex form, let us prove the complex form.

Proof of Fourier: For $m \neq n$, let $k = m - n$,

$$\int_0^L e^{i\frac{2\pi(m-n)x}{L}}dx = \int_0^L \cos\left(\frac{2\pi kx}{L}\right) + i\sin\left(\frac{2\pi kx}{L}\right)dx = 0$$

QED

Fourier Transform

it doesn't limit to periodic functions

$$f(x) = \int_{-\infty}^{\infty}\hat{F}(p)e^{ipx}dp$$

and

$$\hat{F}(p) = \frac{1}{2\pi}\int_{-\infty}^{\infty}f(x)e^{-ipx}dx$$

Proof:
We need delta distribution

$$\delta(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty}e^{ipx}dx = \begin{cases} 0 & x \neq 0 \\ \infty & x = 0 \end{cases}$$

It has property that

$$\int_{-\infty}^{\infty}f(x)\delta(x)dx = f(0)$$

Now back to prove Fourier, substitute $\hat{F}(p)$ into $f(x)$, we get

$$\iint_{-\infty}^{\infty} f(y)e^{ip(x-y)}\,dpdy = \int_{-\infty}^{\infty} f(y)\delta(x-y)dy = f(x)$$

QED

Compare the proof of Fourier series and Fourier transform, we conclude they are two different animals.

Discrete Fourier transform

It is named "transform", but in my opinion, it is misnamed. It is more close to Fourier series than to Fourier transform. I guess it is named because of its applications in solving PDEs. We use it like Fourier transform.

$$f(x) = \sum_{n=0}^{N-1} \hat{F}(n)e^{i\frac{2\pi nx}{N}}$$

and

$$\hat{F}(n) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)e^{-i\frac{2\pi nx}{N}}$$

It is Fourier series with replacements of

$$\frac{x}{L} \to \frac{x}{N} \text{ and } \frac{dx}{L} \to \frac{1}{N}$$

That is because now $x$ is integer. What about its proof?

Proof: First notice any non-zero integer $k \in [-(N-1),(N-1)]\backslash\{0\}$. Roots of unity or just Geometric series.

$$\sum_{x\in[0,N-1]} e^{i\frac{2\pi kx}{N}} = \frac{1-e^{i\frac{2\pi kN}{N}}}{1-e^{i\frac{2\pi k}{N}}} = \frac{1-1}{} = 0$$

Now substitute $f(x)$ into $\hat{F}(m)$, $m \in [0, N-1]$.

$$\hat{F}(m) = \frac{1}{N}\sum_{x,n=0}^{N-1} \hat{F}(n)e^{i\frac{2\pi(n-m)x}{N}} = \underbrace{\frac{1}{N}\sum_{\substack{x\in[0,N-1]\\ n=m}} \hat{F}(m)}_{\hat{F}(m)} + \frac{1}{N}\sum_{n\in[0,N-1]/\{m\}} \hat{F}(n) \underbrace{\sum_{x\in[0,N-1]} e^{i\frac{2\pi(n-m)x}{N}}}_{0}$$

QED

The proof provides an important inspiration for Fast Fourier Transform (FFT)

Fast Fourier Transform (FFT)

DFT says to Fourier $f(x)$

$$\hat{F}(n) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)e^{-i\frac{2\pi nx}{N}}$$

Blindly we see each $\hat{F}(n)$ takes $N$ products so the complete Transform will be

$$O(N^2)$$

Divide and conquer, split $x$ into even and odd

$$\hat{F}(n) = \frac{1}{N}\underbrace{\sum_{x=0}^{\frac{N}{2}-1} f(2x)e^{-i\frac{2\pi nx}{N/2}}}_{\substack{Fourier\ of\ f(2x)\\ with\ half\ of\ mesh\ size}} + e^{-i\frac{2\pi n}{N}}\frac{1}{N}\underbrace{\sum_{x=0}^{\frac{N}{2}-1} f(2x+1)e^{-i\frac{2\pi nx}{N/2}}}_{\substack{Fourier\ of\ f(2x+1)\\ with\ half\ of\ mesh\ size}}$$

Now we observe that

$$\hat{F}\left(n+\frac{N}{2}\right) = \frac{1}{N}\underbrace{\sum_{x=0}^{\frac{N}{2}-1} f(2x)e^{-i\frac{2\pi nx}{N/2}}}_{same\ as\ above} - e^{-i\frac{2\pi n}{N}}\frac{1}{N}\underbrace{\sum_{x=0}^{\frac{N}{2}-1} f(2x+1)e^{-i\frac{2\pi nx}{N/2}}}_{same\ as\ above}$$

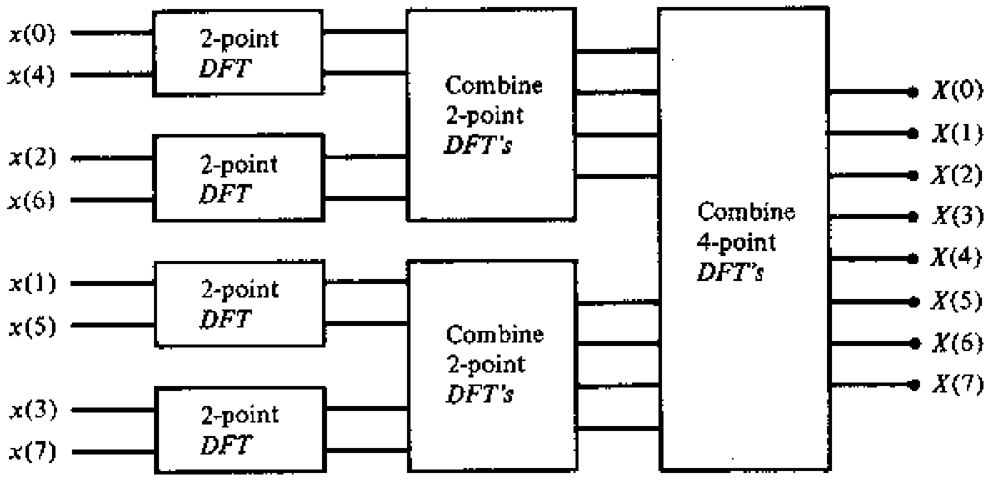
Splitting cuts works to half,

$$T(N) = T\left(\frac{N}{2}\right) + O(1) \rightarrow T = O(\log N)$$

Hence the complete transformation will be

$$T(N) = O(N\log N)$$

```
FFT(f, N)   //first time call pass in f[0],f[1],…,f[N-1]

     if N == 1, return f[0]

     (F_even) = FFT(f[even], N/2)
     //pass in even indices, return N/2 corresponding Fourier Coefficients

     (F_odd)  = FFT(f[odd], N/2)
     //pass in odd indices

     let ANS = empty array size N
     phase = $e^{-i\frac{2\pi}{N}}$

     for n = 0 to N/2-1
            ANS[n] = F_even[n] + (phase^n) * F_odd[n]
            ANS[n + N/2]  = F_even[n] – (phase^n) * F_odd[n]

     Return ANS
```

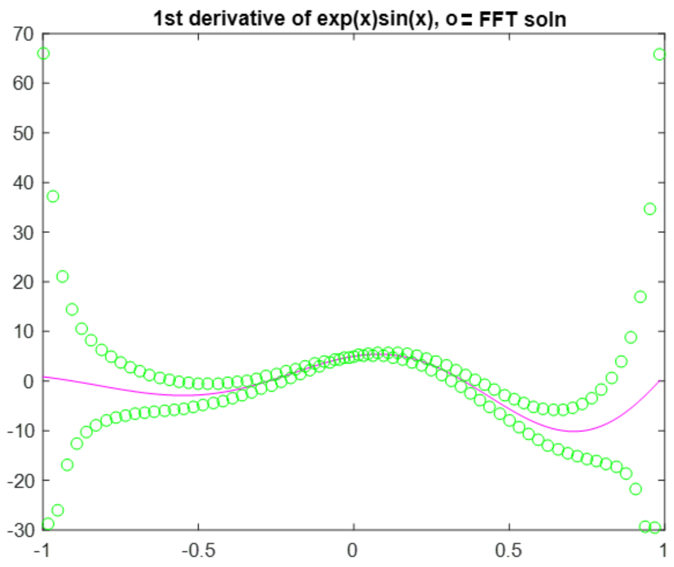


This diagram shows the three stages of computation and points dependence. Picture from http://cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html

What if the function is not periodic at boundary?

```
L= 2;  n=128;

x2 = linspace(-L/2, L/2, n+1);  x= x2(1:n);

u=exp(x).*sin(5*x);
ud=exp(x).*sin(5*x)+5*exp(x).*cos(5*x);
```

```
k=(2*pi/L)*[0:(n/2-1) (-n/2):-1];
ut = fft(u);  udt = i*k.*ut;  udtt = ifft(udt);
//here we applied convenient derivative

plot(x, ud,'m', x, udtt,'go')
title('1st derivative of exp(x)sin(x), o = FFT soln')
```



Bad at boundary, b/c it tries to do periodic extension,
which is called Gibbs phenomenon. Fourier solution bounds up and down.

Next we study Chebyshev transform, which is very gentile at boundary.

- Fast Chebyshev Transform

$$O(N \log N)$$

<u>Orthogonal polynomials</u>

Let

$$\langle f, g\rangle = \int_{x_1}^{x_2} f(x)g(x)W(x)dx$$

be weighted inner product with $w \geq 0$ a.e. Then $(f,g)_w = 0 \Leftrightarrow$ they are orthogonal

Gram-Schmidt tells us that one can get an orthonormalized basis of polynomials by starting with $1, x, x^2, \ldots$

- Legendre polynomials

By letting $w = 1, x_1 = -1, x_2 = 1$

- Chebyshev polynomials

By letting $w = \frac{1}{\sqrt{1-x^2}}, x_1 = -1, x_2 = 1$

$$T_n(\cos\theta) = \cos n\theta$$

Gives uneven grids.

- Hermite polynomials

By letting $w = e^{-x^2}, x_1 = -\infty, x_2 = \infty$

- Bessel polynomials

By letting $w = x, x_1 = 0, x_2 = a$

$$\int_0^a x J_p(j_{pn}\frac{x}{a}) J_p(j_{pm}\frac{x}{a})\, dx = \frac{a^2}{2}\left[J_{p+1}(j_{pn})\right]^2 \delta_{n,m}$$

Furthermore, let norm $\|f\|_w^2 = (f, f)_w$. Then the best polynomial estimate $p^*$ of a given function $f$ is

$$\|f - p^*\|_w^2 \leq \|f - p\|_w^2 \ \forall\, p \in P$$

and clearly if use (any of the above) orthogonal basis $\{\phi_i\}$, then

$$p^* = \sum (f, \phi_i)_w \phi_i$$

<u>Chebyshev derivative matrix</u>

$$D_N = \left|\begin{array}{c|ccc|c} \frac{2N^2+1}{6} & & 2\frac{(-1)^j}{1-x_j} & & \frac{1}{2}(-1)^N \\ \hline & & & \frac{(-1)^{i+j}}{x_i-x_j} & \\ -\frac{1}{2}\frac{(-1)^i}{1-x_i} & & \frac{-x_j}{2(1-x_j^2)} & & \frac{1}{2}\frac{(-1)^{N+i}}{1+x_i} \\ & \frac{(-1)^{i+j}}{x_i-x_j} & & & \\ \hline -\frac{1}{2}(-1)^N & & -2\frac{(-1)^{N+j}}{1+x_j} & & -\frac{2N^2+1}{6} \end{array}\right|$$

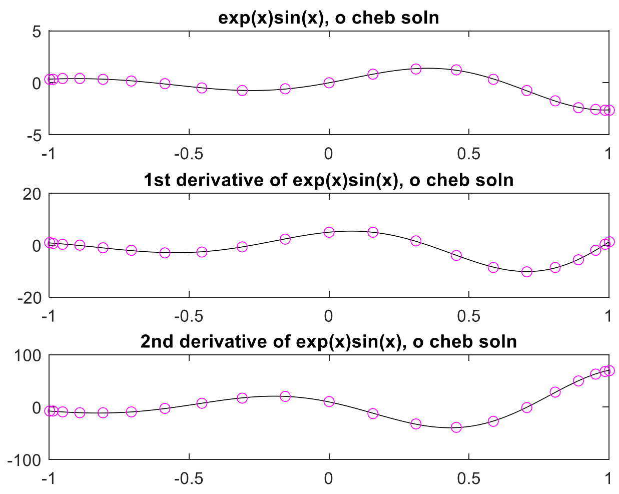
where $x_j = \cos(\frac{j\pi}{N}), j = 0, \ldots, N$

http://ocw.mit.edu/courses/mathematics/18-336-numerical-methods-for-partial-differential-equations-spring-2009/lecture-notes/MIT18_336S09_lec9.pdf

Code was copied from Nathan Kutz http://courses.washington.edu/amath581/581.pdf page 105-108

```
x=-1:0.01:1;
u=exp(x).*sin(5*x);
ux=exp(x).*sin(5*x)+5*exp(x).*cos(5*x);
uxx=-24*exp(x).*sin(5*x)+10*exp(x).*cos(5*x);

N=20;

[D,x2]=cheb(N); %this return x2 \in [-1,1]
u2=exp(x2).*sin(5*x2);
u2x=D*u2;
u2xx=D*u2x;

subplot(3,1,1)
plot(x,u,'k-',x2,u2, 'mo')
title('exp(x)sin(x), o cheb soln')

subplot(3,1,2)
plot(x,ux,'k-',x2,u2x, 'mo')
title('1st derivative of exp(x)sin(x), o cheb soln')

subplot(3,1,3)
plot(x,uxx,'k-',x2,u2xx, 'mo')
title('2nd derivative of exp(x)sin(x), o cheb soln')

%cheb.m%

function [D,x]=cheb(N)
if N<=0, D=0; x=1; return; end
x=cos(pi*(0:N)/N)';
c=[2; ones(N-1,1); 2].*(-1).^(0:N)';
```

```
X=repmat(x,1,N+1);
dX=X-X';
D=(c*(1./c)')./(dX+eye(N+1));
D=D-diag(sum(D'));
```



$10^{-8}$ accurate, cheb points do good job at boundary solves polynomial wiggle problem

- Fast Multipole Method (FMM)

Invent by Leslie Greengard (Former-NYU-Courant director) and Vladimir Rokhlin

A thorough bibliography https://web.njit.edu/~jiang/math707.html

The idea was borrowed from multipole expansion, monopole ($\sim 1/r$ ), dipole ($1/r^2$), … from electromagnetism.

Recall we showed the solution to heat equation

$$u(x,t)=\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty}\underbrace{u(y-x,0)}_{kernel}\underbrace{\frac{e^{-\frac{1}{2}\frac{y^2}{(\sqrt{2t})^2}}}{\sqrt{2t}}}_{weight}dy$$

This convolution form is very general as we proved it was from Fourier of product of algebraic equations. Many physical systems have this form.

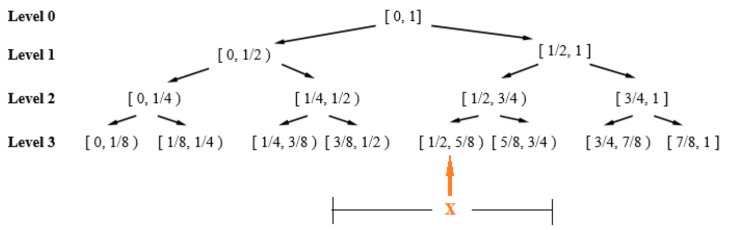
For numerical simulation (or numerical animation if allow $t$ to grow), one will probably create a 2d heat map (assume above $x, y$ are 2d), and compute

$$u(x,t) = \sum_y K(x,y)\, w(y,t)$$

for every $x$ on the grid. Then just like in DFT, blindly one has to do

$$O(N^2)$$

The idea of FMM is to play with the kernel. Because by physical intuition, if $y - x \gg 1$, the value of $u(y-x, 0)$ should not vary too much from the point of view of $x$, when computing $u(x,t) = \sum_y K(x,y)\, w(y,t)$, we change the number of grids, the farther we go, the less grids are there. In fact we organize the grids like a binary tree



Picture from http://math.nyu.edu/faculty/greengar/shortcourse_fmm.pdf page 8

For example we want to find $u(x = 0.55, t)$, then we compute the sum, for $y$ points in [3/8, 3/4], we do one by one counts. For $y$ points in [1/4, 3/8], $y$ ∈[4/3, 7/8], [7/8,1] are treated as one point, and points in [0, 1/4] are created as one point. So we reduce $N$ sums to $\log N$ sums, thus

$$O(N \log N)$$

The initial construction of the binary tree is the key. It defines the accuracy of the computation.

- Other Transforms

Zak transform

$$Z_a[f](t,w) = \sqrt{a} \sum_{k=-\infty}^{\infty} f(at + ak) e^{-2\pi k w i}$$

Wigner Ville Distribution

$$W_x(t,\omega) = \int_{-\infty}^{\infty} x(t + \frac{\tau}{2}) x^*(t - \frac{\tau}{2}) \mathrm{e}^{-i\omega\tau} d\tau$$

Wavelets

Mexican hat

$$\psi(t) = \frac{2}{\sqrt{3\sigma}\pi^{\frac{1}{4}}} \left(1 - \frac{t^2}{\sigma^2}\right) e^{\frac{-t^2}{2\sigma^2}}$$

http://mathworks.com/help/wavelet/ref/wfilters.html?refresh=true#zmw57dd0e43152

$$[W_\psi f](a,b) = \frac{1}{\sqrt{|a|}} \int_{-\infty}^{\infty} \overline{\psi\left(\frac{x-b}{a}\right)} f(x)dx$$,

*a* = dilation, *b* = translation

## The Present: High Performance Computing

### 5. PRAM Model

- Brent's Theorem

Given $P$ processors, $n$ data, think $W(n)$ total number of work per data point as the total number nodes of a directed acyclic graph (DAG) and span, $D(n),=$ the longest (critical) path, the unit time to execute

$$\frac{W-D}{P} + D \geq T_p(n) \geq \max\left\{D(n), \left\lceil\frac{W(n)}{P}\right\rceil\right\}$$

Above assume each work takes same $O(1)$ time.

Proof of the upper bound:
Breaking $W$ to $D$ different phases, the critical path will pass through the phases and leaving one critical node and $W_k$ concurrent works, i.e. $\sum W_k = W$. That is

$$T_p(n) = \sum_{k=1}^{D(n)} \left\lceil\frac{W_k(n)}{P}\right\rceil \leq \sum_{k=1}^{D(n)} \frac{W_k(n)-1}{P} + 1 = \frac{W-D}{P} + D$$

Define

$$level\ of\ parallelism = \frac{W(n)}{D(n)}$$

In terms of directed acyclic tree, assume all process

The speedup achieved

$$S_P(n) = \frac{best\ sequential\ time\ T_s(n)}{T_p(n)}$$

If we want $S_P(n)$ to be linear in $P$, we get Amdahl's law

$$\frac{T_s}{T_P} \leq \frac{T_s}{\frac{W-D}{P} + D} = \frac{P}{\frac{W}{T_s} + \frac{P-1}{T_s/D}}$$

Hence we want the denominator to be order 1, which implies

Work-optimality

$$W = O(T_s)$$

Weak-scalability

$$P = O\left(\frac{T_s}{D}\right)$$

First taste of parallel algorithm

```
def reduceSum(list)

        ‘’’ find sum of list ‘’’

        if len(list) == 1: return list[0]
        A = spawn reduceSum(list[ : n / 2])
        B = reduceSum(list[ n / 2 + 1 : ])
        sync // wait spawn finishes, then do sum
        return A + B
```

A Library of Parallel Algorithms, http://cs.cmu.edu/~scandal/nesl/algorithms.html
Parallel Algorithms from Microsoft, https://msdn.microsoft.com/en-us/library/dd470426.aspx

- Parallel Loop

```
def parFor(foo, a, b)

      ‘’’  foo is some function. Argument of foo is current iterator,
      ‘’’  a, b are starting and ending range of the cursor.

       let n = b – a + 1
       if n == 0: return foo(a)
       let m == a + n / 2
       spawn parFor(foo, a, m-1)    //split in half
       parFor(foo, m, b)            //the second half
       sync
```

Above algorithm has

$$D(n) = O(\log n)$$

Notice we don’t do this

```
for i in range(a,b + 1):
        spawn foo(i)
```

Because above serial for-loop will still push `foo` to the stacks $n$ times, i.e. assume the time complexity of `foo` is $< O(n)$, then overall span is still

$$D(n) = O(n)$$

not good.

Example

$$y = y + Lx$$

```
parFor i = range(n)
      t = [0]*n
      parFor j = range(n)
            t[i] = L[i,j] * x[j]
     //after loop sync is implicitly called
      y[i] = y[i] + reduceSum (t)
```

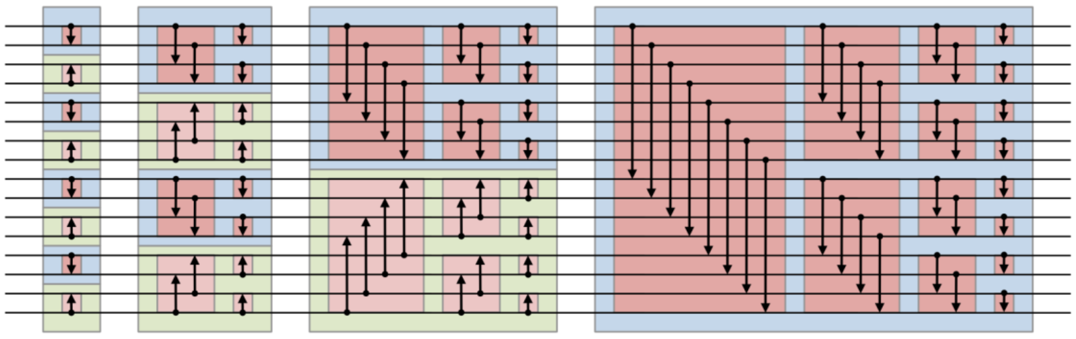
The span of above is $D(n) = O(\log n)$

We can simplify above. It works the same because Python or Fortran will automatically invoke parallelism for pair multiplication when index slicing/bundle notation is involved,

```
parFor i = range(n)
      y[i] = y[i] + reduceSum (L[i,:] * x[:])
```

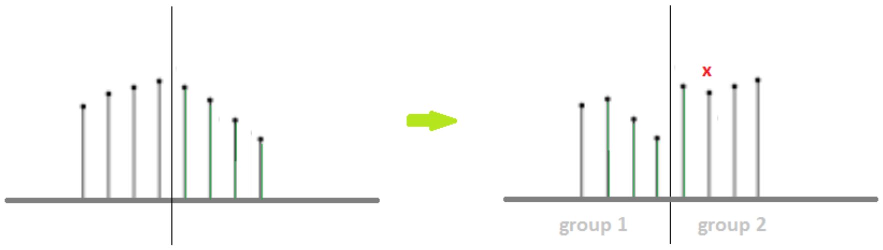
- Parallel Sort

Bitonic Sorting


Picture curtesy https://en.wikipedia.org/wiki/Bitonic_sorter

A bitonic sequence is increasing than decreasing, having larger values clustering in the middle and smaller values by the sides. Read above graph from left to right. The arrows are comparators. Up arrows indicated that sending smaller values on top; downward arrows indicated putting larger values on top. In the first column we applied arrows to 8 pairs to data, we got 4 4-element bitonic sequences. Next we applied sorting to each 4-element bitonic sequence in 2 steps. First compared 1st and 3rd, and compared 2nd and 4th. This resulted the bitonic sequence to split into two groups. Every element of one group was larger than the other group. In the second step we sorted the two groups, so we got 2 8-element bitonic sequences. Next we applied sorting to each 8-element bitonic sequence in 3 steps, and we merged them into 1 16-element bitonic sequence, then we applied sorting to it.

Proof of the split algorithm resulting two separated groups, one of which is larger than the other.



After splitting, we pick an arbitrary element $x$ from group 2. We show $x >$ group 1. Clearly $x$ is larger than its counter part 2nd element in group 1. $x$ is also larger than any element to the left of the 2nd element in group 1 because of the following.

Case 1 suppose before splitting, $x$ was in group 2. So the original 2nd element was in group 1, then any element left to the 2nd element was originally smaller than $x$, hence they were smaller than any elements to the left of $x$ from group 2, so none of them would get switched during splitting, so they are still smaller than $x$.

Case 2, suppose $x$ was in group 1. Then $x$ was larger than any element to its left in the original group 1. After splitting, those elements can only get smaller, so they are still smaller than $x$.

Similarly, one can prove that $x$ is larger than any element to the right of 2nd element in group 1. QED

The serial complexity is not impressive

$$D(n) = O(\log^2 n), W(n) = O(n \log^2 n)$$

However its parallel implementation is impressive.

Parallel Algorithm for Bitonic Sorting

```
def bitonicSplit(A[0 : n - 1])
```

```
       //assume 2 | n
        parFor i = range(n / 2)
               a = A[i]
               b = A[i + n / 2]
               (A[i], A[i + n / 2]) = (a , b)

 def bitonicMerge(A[0 : n - 1])
        if n >= 2 //assume 2 | n
               bitonicSplit(A[:])
               bitonicMerge(A[0 : n / 2 - 1])
               bitonicMerge(A[n / 2 : n - 1])

 def generateBitonic(A[0 : n - 1])
        if n >= 2 // assume 2 | n
               spawn generateBitonic(A[0 : n / 2 - 1])
               generateBitonic(A[n / 2 : n - 1])
               sync
               spawn bitonicMerge(A[0 : n / 2 - 1])
               bitonicMerge(A[n / 2 : n - 1])

 def bitonicSort(A[0 : n - 1])
        generateBitonic(A[:])
        bitonicMerge(A[:])
```

Parallel QuickSort

```
def QS(A[0 : n - 1])
        if n == 1:  return A[0]
        pivot = A[random.randrange(0 , n)]
        L = A[A[:] <= pivot]
        R = A[A[:] > pivot]
        AL = spawn QS(L)
        AR = QS(R)
             sync
        return AL + AR
```

In fact Fortran, python will perform parallelism behind

L = A[A[:] <= pivot],   R = A[A[:] > pivot]

Call Parallel Scan

$$D(n) = O(\log^2 n), W(n) = O(2n)$$

```
F[0 : n-1] = [0] * n
F[:] = (A[:] <= pivot)   //this is clearly parallel
K[:] = addScan(F[:]) //cumulative sum parallel
```

```
m = K[n - 1]
L[0 : m - 1] = [0] * m
parFor i = range(n)
          if F[i] == 1:  L[K[i]] = A[i]
//finish L

def addScan(A[0 , n - 1])  //assume n = 2^a
           if n == 1:  return A[0]
           IO[0 , n / 2 - 1] = range(1, n , 2)  //odd indices
           IE[0 , n / 2 - 1] = range(0, n , 2)  //even indices
           A[IO] = A[IE] + A[IO] // pairwise additions 1st + 2nd; 3rd + 4th
elements parallel
           A[IO] = addScan(A[IO])
           A[IE] = A[IE[1:]] + A[IO[1:]]
```

- Parallel Linked List

To parallelize it, we need random access. First we hash and store linked list to an array pool.

Let $V[\quad]$ = array of the value of the linked list and the indices of $V$ are the keys. It may contain unused cells. Remember the index position of the head.
Let $N[\quad]$ = a parallel array of $V[\quad]$. Its indices are in the same order of $V[\quad]$ and its values are the key to the next node of the linked list points to. For unused cells of $V[\quad]$, their corresponding values in $N[\quad]$ should set to be NULL.

For example, suppose the current pointer of the linked list points to address whose hash key is $i = 13$. Then its value in the linked list is

$$V[13]$$

The next node has key $N[13]$, so the value of the next node is

$$V[N[13]]$$

Linked List Rank Algorithm (Wyllie)

First set every node to 1, then use parallel cumulative sum, which follows the same idea in addScan for parallel quicksort.

To do parallel sum, first do 2 pair neighbor sum, than use jump to connect the node to its previous-previous node and add it to the node, hence it's the sum of 4 consecutive neighbors, then jump again to its previous-previous-previous-previous and add it to the node, and etc.

With a little twist, we can get parallel access to say, the 100th node. Just stop the program when rank reaches 100.

```
def updateRanks(Rin, Rout, N)
       // 2-pair neighbor sum in parallel
       parFor i in range(n):
               if N[i] != NULL: Rout[N[i]] = Rin[i] + Rin[N[i]]

def jumpList(Nin, Nout):
       // jump to next next node
       parFor i in range(n)
               if Nin[i] != NULL: Nout[i] = Nin[Nin[i]]

def swap(A,B)
       (A,B) = (B,A)


def rankList(V[], N[], h) //h=head
       let R1[0 : n - 1] = R2[0 : n - 1] = [1] * n
       let N1[0 : n - 1] = N2[0 : n - 1] = N[]  //make two copies

       for i in range(log(n) + 1):
               updateRanks(R1,R2,N1) // R2 gets modified
               if i == log(n):  return R2
               jumpList(N1,N2)       // N2 gets modified
               swap(R1,R2)
               swap(N1,N2)
```

The work and span of parallel linked list algorithm is

$$W(n) = O(n \log n), D(n) = O(\log n)$$

Its works are not linear as in serial case. There is a tradeoff between work-optimal and run-time-optimal. There is a so called work-optimal list scan algorithm that will reduce $W$ to

$$W(n) = n \log \log n$$

<u>Work-optimal Linked List Rank Algo</u>

- shrink the list to size $m = O\left(\frac{n}{\log n}\right)$ via independent set
    - find just an independent set, not the max indep set!
    - an independent set is a set that no two elements are directly linked together
    - remove each element in the independent set from the list and add its current rank to its successor. So basically we preprocess the list and do some cumulating sum on the fly. Everything will be in parallel. That is why we require independent sets, not to remove two linked nodes all together in one concurrent time.
    - keep doing that till the list become size $O\left(\frac{n}{\log n}\right)$
- run Wyllie on the shrinked list $W(n) = O(m \log m) \sim O(n)$
- restore the full list.

- Clearly those removed elements don't have ranks. To restore them, we will run backward of the removal steps. Get back their ranks from their remaining successors.

How to parallel find an independent set?

```
def parIndSet(N[:])

       ‘’’ N is defined before, The array of the key of the next node
       ‘’’ return the indices of indep set
       C1 = array of random bool of size len(N)
       C2 = C1  // copy

       parFor i = range(len(N))
                if (C1[i] == True) and (N[i] != NULL) and (C1[N[i]] == True):
                           C2[i] = False

       return C2[i]  // index of True is the independent set.
```

How many times $k$ do we have to run to shrink the list down to $O\left(\frac{n}{\log n}\right)$? About half of them get assigned TRUE and about ¼ of total get assigned to have its next node is also TRUE, so each time ¼ of total becomes independent set, i.e.

$$\left(\frac{3}{4}\right)^k n \approx \frac{n}{\log n} \to k = O(\log\log n)$$

So the total Work of work-optimal linked list algo

$$W = O(n \log\log n)$$

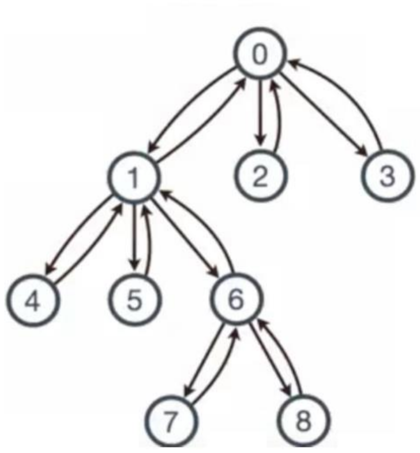
- Parallel Tree

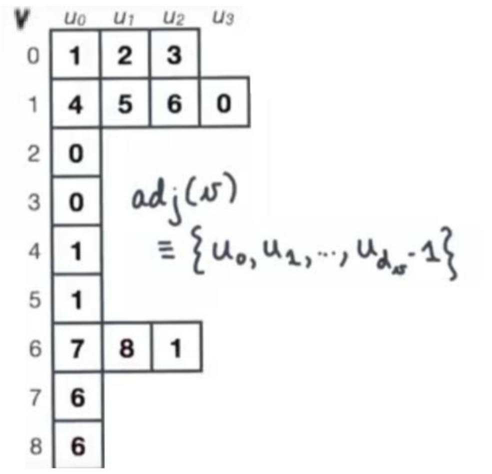
We linearize tree via some traversal and we get Euler Tour Tree

Example: Convert a binary tree to double linked list

```
 node bTreetoDLL(node root):

           ‘’’ in-order traversal

           if root.left == NULL and root.right == NULL
                      return root
           if root.left != NULL
                      pt1 = bTreetoDLL(root.left)
                      root.left = pt1;  pt1.right = root
           if root.right != NULL
```

```
                    pt2 = bTreetoDLL(root.right)
                    root.right = pt2;  pt2.left = root
```





| V | $u_0$ | $u_1$ | $u_2$ | $u_3$ |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | |
| 1 | 4 | 5 | 6 | 0 |
| 2 | 0 | | | |
| 3 | 0 | | | |
| 4 | 1 | | | |
| 5 | 1 | | | |
| 6 | 7 | 8 | 1 | |
| 7 | 6 | | | |
| 8 | 6 | | | |

$adj(v) \equiv \{u_0, u_1, \ldots, u_{d_v}-1\}$

Picture from Rich Vuduc, *High Performance Computing*, Udacity lecture 1-5

From edge $(0,1) \to (1,4) \to (4,1) \to (1,5) \to (5,1) \to (1,6) \to (6,7) \to (7,6) \to (6,8) \to (8,6) \to (6,1) \to (1,0) \to (0,2) \to (2,0) \to (0,3) \to (3,0) \to (0,1)$

The map is

$$S(U_i(V), V) = (V, U_{(1+i) mod\, d_V}(V))$$

Where $V$'s are the vertices, and $U_i(V)$'s are the immediate neighbors of $V$, enumerated by $i$.

Proof

If $d_V = 1$, like node 4, then going into vertex 4 has to come out of vertex 4, i.e.

$$S(1,4) = (4,1)$$

Indeed

$$S(U_0, V) = (V, U_{1\, mod\, 1}) = (V, U_0)$$

If $d_V > 1$, then $(1 + i)\, mod\, d_V$ will rotate all its children and in the end it will return back to the first path going into the vertex. QED

With the map, we can create linear circular linked list in parallel out of tree, thus the parallel algorithms for linked list become applicable.

- Parallel Graph

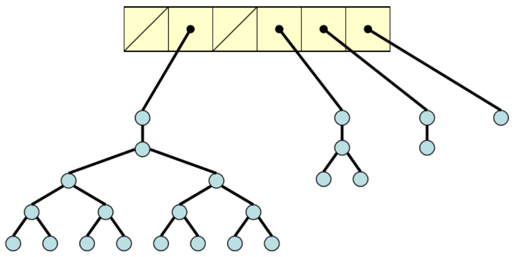
We know sequential BSF

$$W = O(|V| + |E|)$$

Now we do parallel. Recall in sequential, we used a queue to pipe nodes. Now we want to use one list for each depth $d$, within each depth we can do parallel and cross visits should not be an issue.

```
def level_sync_BSF(G = (V,E), s) //s = starting vertex

       ‘’’ along the way, we find the depth of node ‘’’

       D[V-s] = -1; //label of the depth of each node
       F[0] = [s]; // first level
       d = 0
       while F[d] not empty
             F[d+1] = []
             processLevel(G, F[d], F[d+1], D)
             d++
       return D
```

We will use binomial heap as bag data structure, but we don’t need to force smaller numbers on top as required by the heap. The reason we use such bag structure is because adding and splitting are $O(\log n)$. Recall binomial heap is an array with pointers to pennants.



Picture from http://courses.csail.mit.edu/6.884/spring10/labs/lab4.pdf page 4

Adding two pennants of same size or splitting a pennant to two equal size is $O(1)$. Therefore splitting a bag is just to split pennants and travel down the array to find non-occupied cells.

```
def processLevel(G=(V,E), F[d], F[d+1], D):
        if |F[d]| > cutoff
                (A,B) = bigSplit(F[d])    // log(n)
                spawn processLevel(G, A, F[d+1], D)
                processLevel(G, B, F[d+1], D)
                sync
        else
                for v in F[d]
                         parFor (v,w) in E
                               if D[w] = -1 then
                                          D[w] = d + 1
                                          bagAdd(w)    //log(n)
```

So the total span

$$D(n) = O(d \log^r(|V| + |E|))$$

## 6. Distributed Memory Model

- Message-Passing Model

It is called because all nodes' memories are private. Sharing data have to be sent over networks. Each node knows how many nodes are there in the network and everybody's ID (rank).

<u>We assume</u>

- the networks are fully connected, for now ignoring the network topology, assume sending or receiving between any two nodes takes same amount of time.
- links are bidirectional; allow sent and receive
- Each node can do only one send + one receive at a time
- cost to send or receive $n$ words

$$T_{msg}(n) = \alpha + \beta n$$

where $\alpha$ = latency, typical $10^{-6} sec$, $\beta$ = 1 / bandwidth, which has the unit of time / words, typical $10^{-9} sec$, compared to operation time $10^{-12} sec$ (Typical CPU 3.5GHz, i.e. it times $3.5 \times 10^9$ clock cycle in 1 sec, 4 cores, 64 floating points. Altogether it gives $1.1 \times 10^{-12}$ sec/per operation -- Mind-blowing!).

- if there are $k$ messages are using the same segments of the network, called $k$-way congestion

$$T_{msg}(n) = \alpha + k\beta n$$

How are two nodes sending or receiving?
Called point-to-point communication, later we will do one-to-all and all-to-all communications.

<u>Single Program Multiple Data (SPMD)</u>

2-sided messaging: node A sends stuff to node B

Machine A

```
 // create a separated event to listen
 handle <- sendAsync(buffer[1:n], B)
         //buffer are the content to be send; A knows B's rank

… //do other stuff
```

```
 wait(handle)  //or blocking wait till handle completes

//The handle will not complete if there is not a B out there awaiting for the
//message. Depend on the implementation if it returns handled event, it means
//most likely B has received, not 100%, but for sure the buffer is cleared.
```

Machine B needs to have a receiving event. For a signal to go through both ends of the wire have to open.

```

handle <- recvAsync(buffer[1:n], A)
         //buffer are the content for receive to write to

  //if handle return handled event, it means B has received.
```

- Distributed Operations

First taste of distributed computation: All-to-one reduceSum

Run the follow code to all machines

```

 ‘’’ add values of all nodes and store the sum to rank = 0 node ‘’’
 ‘’’ s = the value to sum
 ‘’’ P = total nodes
 ‘’’ rank = its own id

 bitmask <- 1
 while bitmask < P
         Partner <- rank ^ bitmask  //bitwise XOR
          //when bitmask = 0000000 $\underbrace{1}_{b^{th}}$ 0000,
          //XOR connects rank and its partner have same bits except the $b^{th}$

         if rank & bitmask //bitwise AND, so only nodes that has 1 on $b^{th}$
                    sendAync(s, Partner)
                    wait(*)
                    break   //once it sent, its task is done

         elif (Partner < P)
                    receAync(t, Partner)
                    wait(*)
                    s += t
         bitmask <- (bitmask << 1)  //bitwise shift, basically *2

  if rank == 0
         print “DADA: ” + s

```

<u>All-to-One Reduce</u>

More general if $s$ is vector of size $n$, sum component-wise using above algorithm, time is

$$T(n) = (\alpha + \beta n) \log P$$

This operation reduces information from all nodes to one node. Its standard API is

$$reduce(A_{local}[1{:}n], root)$$

Each node calls $reduce$ and each node except the root send its $A$ to $root$.

<u>One-to-All Broadcast</u>

Copy the same $n$ data from one node to all nodes. Its standard API is

$$broadcast(A_{local}[1{:}n], root)$$

Each node calls $broadcast$ and only root does the sending.

Implementation of One-to-All Broadcast

Root (suppose its rank 0) sends $A$ to rank 1, then rank 0, 1 send to 2, 3 nodes, then 0-4 nodes send to 5-8 nodes. Thus

$$T(n) = (\alpha + \beta n) \log P$$

<u>All-to-One Gather</u>

Its standard API is

$$gather(In[1{:}m], root, Out[1{:}m][1{:}P])$$

Each nodes calls $gather$ and each except the root sends its local In buffer, array of data size $m$ to the root. The $Out[1{:}m][1{:}P]$ only works for the root. The root will receive the messages from each node and store them into the 2 dimensional array, Out buffer. The total size of the transmitted data is

$$n = mP$$

<u>One-to-All Scatter</u>

Reverse of <u>All-to-One Gather</u>

$$scatter(In[1{:}m][1{:}P], root, Out[1{:}m])$$

<u>All-Gather</u>

$$allGather(In[1:m], Out[1:m][1:P])$$

will do part of All-to-One Gather and duplicate the root to all notes. It is kind like All-to-One Gather followed by One-to-All Broadcast.

```
gather(In, out, root) //root is the receiver
broadcast(reshape(Out), root) //root is the sender
      //reshape is changing 2-d array to 1-d
      //reshape is only a logic operation, because in C, matlab 2-d array
      //is physically stored as 1-d array, so the operation is $O(1)$.
```

Reduced Scatter

Reverse of All-Gather

$$reduceScatter(In[1:m][1:P], Out[1:m])$$

Lower Bound

We see that above operations have one thing in common. There is at least one node who either receive or send message from or to all other nodes. Since it can only receive or send one piece of data at a time so the latency is in its best to be in parallel, so $\alpha \log P + \beta n$ is the best we can do. This gives the lower bound

$$T(n) = \Omega(\alpha \log P + \beta n)$$

where $n$ is total data being received or sent. We now show one-to-all scatter attain the minimum.

Just like the implementation of broadcast. Root (rank = 0) sends $In[1:m][1:P/2]$ to rank 1 node. Then in parallel 0 node sends $In[1:m][P/2+1:3P/4]$ to rank $P/2$ and rank 1 node sends its half of the $In$ buffer to some other node,

$$T(n) = \alpha \log P + \beta \sum_{i=1}^{\log P} \frac{n}{2^i} = \alpha \log P + \beta n \left( \frac{1 - \frac{1}{2p}}{1 - \frac{1}{2}} - 1 \right)$$

Notice before we discussed that All-Gather was like All-to-One Gather followed by One-to-All Broadcast, but All-to-One Gather attained the minimum while One-to-All Broadcast did not. So the overall time is still

$$T(n) = (\alpha + \beta n) \log P$$

Let us try a new approach to implement All-Gather, bucketing/pipeline and former approach is like tree.

- each node sends $In[1{:}m]$ to its next node $(rank + 1)\ mod\ P$.
- each node sends the new messages it just received from $(rank - 1)\ mod\ P$ and passes it to $(rank + 1)\ mod\ P$.
- do this for $P$ times.

Overall run time is

$$T(n) = (\alpha + \beta m)P = \alpha P + \beta n$$

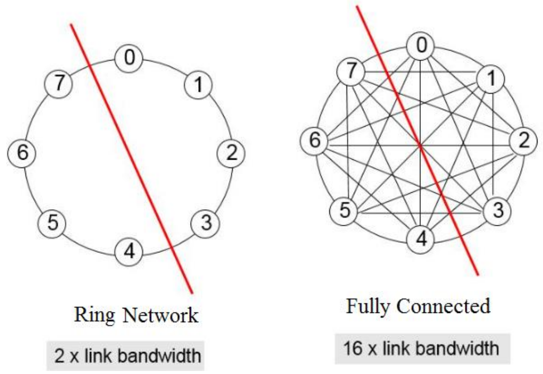
Compare these two approaches, we see that the new approach is bandwidth optimal, because when the message size is very large, the bandwidth term dominates

$$\underbrace{\alpha P}_{\sim 0} + \beta n \ll \underbrace{\alpha \log P}_{\sim 0} + \beta n \log P$$

This also suggests that there may be a bandwidth-optimal approach to One-to-All Broadcast so its run time can get down from $(\alpha + \beta n)\log P$ to $(\alpha P + \beta n)$. Indeed One-to-All Broadcast can be thought of one-to-all scatter followed by All-Gather.

- Distributed Network Topology

Physical networks are not fully connected. So the physical run time is different from the ideal model we discussed above. The one thing characterizes the most is the bisection bandwidth, which is the minimal number of links to be taken out in order to separate the network into two disconnected components. For example



Picture from https://nanohub.org/courses/ECE695R/o1a/asset/11112 page 5

Because most parallel codes are done in splitting processors in half, we see from the two graphs above the ring network bisection bandwidth is 8 times smaller than a full connected (ideal) network, so we should expect for ring network the run-time bandwidth is roughly 8 times slower, $\beta \rightarrow 8\beta$.

Of course it also depends heavily on the algorithm. For example, the All-Gather implementation we discussed before will work equally well for the ring and fully connected networks, because in the All-Gather implementation messages are passed in circles. Each node only talks to the nodes in front and in behind of it.
So to do an algorithm specific analysis of the network, we need to map the logical networks (the routes that are used in passing data) to the physical network. The ratio of the available number of paths to in the logical network to the physical network is called the congestion $k$. E.g. in the graph above if all the cut-out bisection paths are involved in the logic, then $k \geq 8$.

In sum

| | **Diameter** | **Bisection** | **Num of links** |
|---|---|---|---|
| **Full-connected** | $1$ | $\frac{P^2}{4}$ | $\frac{P(P-1)}{2}$ |
| **Complete binary tree** | $2\log\frac{P+1}{2}$ | $1$ | $P-1$ |
| **Ring** | $\frac{P}{2}$ | $2$ | $P$ |
| **Hypercube** | $\log P$ | $\frac{P}{2}$ | $P\log P$ |
| **d-D Tours (d-mesh with connect bdary of the opposite side)** | $\frac{d}{2}P^{\frac{1}{d}}$ | $2P^{\frac{d-1}{d}}$ | $dP$ |
| **Butterfly (e.g. see FFT)** | $\log P$ | $P$ | $P\log P$ |

- Distributed Matrix Multiplication

We want to solve

$$C_{mn} = C_{mn} + A_{mk}B_{kn}$$

Adding C to itself is just because we are doing it in parallel. It is better to create a $C$ at the beginning with all 0 entries as a place holder.

$$c_{ij} = c_{ij} + \sum_{l}^{k} a_{il}b_{lj}$$

First let us review what if it is done in parallel

```
parFor i = range(m)
        parFor j = range(n)
                 T = [0] * k
                 parFor l = range(k)
                           T[l] = A[i,l] * B[l,j]
                 C[i,j] += reduceSum(T)
```

Total work and spin $n \sim m \sim k$

$$W = O(n^3), \qquad D = \log n$$

Now we do it in distributed memory machines

<u>1-D Algorithm</u>

Use circular network. Each node stores $\frac{n}{P}$ rows of the matrice $A, B,$ and $C.$ So for any node, it has the necessary data of $A$ to compute its responsible portion of $C$, but it lacks portion of $B$. We know 1-D network is efficient at circular shuffling. Here is code to be executed on each node

```
Let A0, B0, C0 = local part of A, B, C
Let B1 = temp storage
for i = range(P)
       sendAsync(B0, next_node)
       recvAsync(B1, prev_node)
       C0 += A0 * B0 // part of rows of A * part of column B
       wait(*)
       swap(B0, B1)
```

There are $P$ rounds of communications and during each round each node sends $n * n/P$ data. What about the flop-time for doing "`C0 += A0 * B0`"?

$$\frac{n^3}{P}\tau, \text{where } \tau = \text{"flop", time takes to do one arithmic } (+,*)$$

So total run-time = max of flop-time and communication time

$$T = \max\{\alpha P + \beta n^2, \frac{n^3}{P}\tau\}$$

This algorithm is not good as it looks. First check scalability

$$S_P(n) = \frac{best\ sequential\ time\ T_s(n)}{T_p(n)} = \frac{P\frac{n^3}{P}\tau}{\frac{1}{2}\frac{n^3}{P}\tau + \frac{1}{2}(\alpha P + \beta n^2)} = \frac{2P}{1 + P\frac{\alpha P + \beta n^2}{n^3 \tau}}$$

We want

$$S_P(n) = O(P) \rightarrow n = \Omega(P)$$

That is because if $n \ll P$, then $S_p \ll O(P)$, i.e. increase the number of processors will do very little to the level of parallelism. But $n = \Omega(P)$ is very bad. That means double the number of processors will octuple the matrix size and quadruple the flop time of each node.

What about the memory requirement?

$$4\frac{n^2}{P}$$

4 is because there are 4 local matrices `A0, B0, C0, and B1.`

<u>2-D Algorithm</u>

Use mash network $\sqrt{P} \times \sqrt{P}$. Store $n/\sqrt{P} \times n/\sqrt{P}$ block of matrices $A, B, C$ to each node.

To compute the portion that any node it is responsible for, it needs the entire rows and entire column. So each node simply broadcasts the necessary piece needed to its horizontal and vertical nodes.

```
for i = range(n/s)
    //dividing the matrix into n/s number of either vertical
    //or horizontal strips with width s. s < n/sqrt(P)
       broadcast(A0, [owner])
    //if the node has a portion of the strip, broadcast it,
    //otherwise be ready to receive it
       broadcast(B0, [owner])
       C += A*B
```

Total flop time is still

$$T = \frac{n}{s}\left(\frac{n}{\sqrt{P}}\right)^2 s\tau + \begin{cases} \alpha\frac{n}{s}\log P + \beta\left(\frac{n}{\sqrt{P}}s\frac{n}{s}\right)\log P & \textit{tree based} \\ \alpha\frac{n}{s}P + \beta\frac{n^2}{\sqrt{P}} & \textit{bucket} \end{cases}$$

Similarly we compute the scalability, and we get the isoefficiency function.

$$n_{tree\ base} = \Omega\left(\sqrt{P}\log P\right)$$
$$n_{bucket} = \Omega\left(P^{\frac{5}{6}}\right)$$

What about memory requirement for each node? Local A, B, C and temp for 2 pieces of strips

$$3\frac{n^2}{P} + 2\frac{n}{\sqrt{P}}s$$

<u>A Lower Bound</u>

It turns out that the bucket bandwidth is the minimal bandwidth.

Let’s first count the number of total multiplications operations per processor, which is unescapable to be

$$\frac{n^3}{P}$$

This cannot be done at once, because each node doesn’t have all the data. It will have to communicate and during one communication it can get maximum $M$ data, so roughly $size(A) \sim size(B) \sim size(C) = \theta(M)$. For such sizes it can perform at most

$$\theta\left(M^{\frac{3}{2}}\right) = multiplications$$

Thus there will be $(n^3/P)/M^{3/2}$ communications and every communication acquires $M$ data, so the total bandwidth is

$$\beta M \frac{n^3}{PM^{\frac{3}{2}}}$$

Assume we build the network whose memory is just enough to hold all the matrix data,

$$M = \theta\left(\frac{n^2}{P}\right)$$

Then we get the minimal bandwidth is

$$\beta \frac{n^3}{P\sqrt{M}} = \beta \frac{n^2}{\sqrt{P}}$$

What about the minimal latency? We say there will be $(n^3/P)/M^{3/2}$ communications, so the minimal latency is

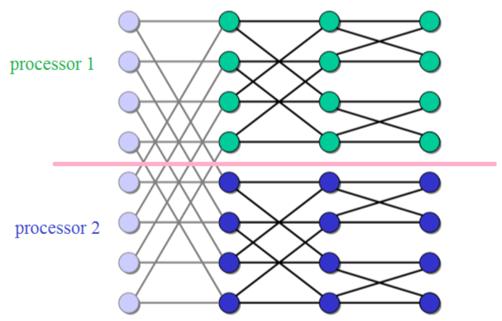
$$\alpha \frac{n^3}{PM^{\frac{3}{2}}} = \alpha\sqrt{P}$$

Cannon Algorithm attains the lower bound.

More recent developments in fast matrix multiplication, see
Grey Ballard, Austin Benson, *A framework for practical parallel fast matrix multiplication*

- Distributed Sort

Distributed Bitonic Merge Sort

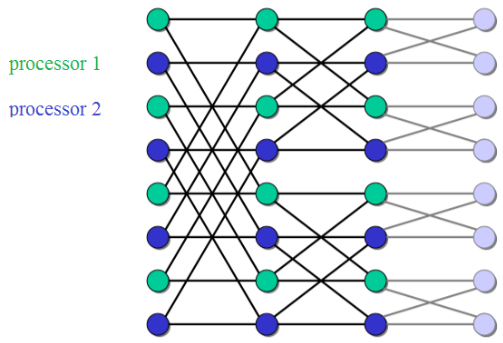


Picture from http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-896-theory-of-parallel-hardware-sma-5511-spring-2004/lecture-notes/interconnection.pdf page 16

Suppose we have a bitonic sequence of 8 elements, and we distribute it to two processors. The nodes in the above graphs are local variables, and the links represent dependencies. Clearly at the initial stage, there will be information exchanging between processor 1, 2, but after the initial stage, no more communications are necessary.

In general, if we distribute $n$ elements in its sequential order to $P$ processors. Then communications will take place in the first $\log P$ steps and followed by $\log n/P$ steps in sorting without communications, so the total communication time is

$$\left(\alpha + \beta \frac{n}{P}\right) \log P$$

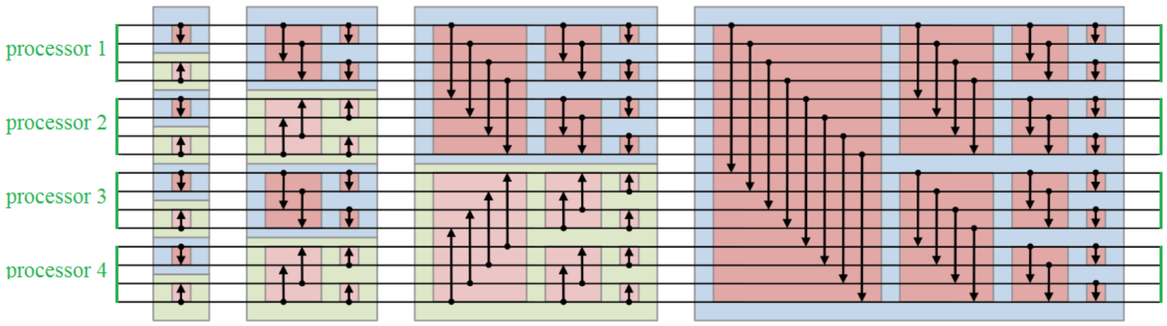


Picture from http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-896-theory-of-parallel-hardware-sma-5511-spring-2004/lecture-notes/interconnection.pdf page 20

However, if we distribute $n$ elements in so-called cyclic order to $P$ processors. Then no communications will take place in the first $\log n/P$ steps and followed by $\log P$ steps in communications, but the total communication time is the same.

## Distributed Bitonic Merge Sort Total Cost

Now we discuss the full bitonic Sort. As far as the previous discussion concern, it only talked about the last stage of the sorting, where a full bitonic sequence had formed.



Picture from https://en.wikipedia.org/wiki/Bitonic_sorter

Suppose there are $P$ processors, and each gets $n/P$ data. There will be $\log n$ number of stages. During each stage $k$, each processor has to do $\frac{n}{P} * k$ number of comparisons, so the total computation time is

$$\tau \sum_{k=1}^{\log n} \frac{n}{P} k = O\left(\tau \frac{n \log^2 n}{P}\right)$$

What about total communication time? Communication starts at stage $\log \frac{n}{P} + 1$, and we apply previous

$$\left(\alpha + \beta \frac{n'}{P'}\right) \log P'$$

with $n' = 2^k$, $P' = 2^{k - \log \frac{n}{P}}$, therefore

$$\sum_{k=\log \frac{n}{P}+1}^{\log n} \left(\alpha + \beta \frac{n'}{P'}\right) \log P' = O\left(\alpha \log P + \beta \frac{n}{P} \log^2 P\right)$$

In sum

$$T(n) = O\left(\tau \frac{n \log^2 n}{P} + \alpha \log P + \beta \frac{n}{P} \log^2 P\right)$$

<u>Distributed Bucket Sort & Sample Sort</u>

Bucket sort has $T = O(n)$, but it assumes the data is uniformly distributed within some range. Suppose there are $n$ data range from $a$ to $b$, and there are $P$ processors. Initially each processor stores $n/P$ data. Let processor $k$ be the $k$th bucket which contains data range from $\left[a + \frac{b-a}{P} k, a + \frac{b-a}{P}(k+1)\right)$. First each processor will do $n/P$ number of comparisons and send on average $n/P^2$ data to one of the other processor. After all communications have completed, then each processor does a local sort. Therefore the total run-time is

$$T(n) = O\left(\tau \frac{n}{P} + \left(\alpha + \beta \frac{n}{P^2}\right) P + \tau \frac{n}{P} \log \frac{n}{P}\right)$$

which is $\sim O(n)$, if $P = \theta(n)$.

Bucket sorting has serious limitation, because it assumes the data set is uniformly distributed. To correct it, sample sort was invented. First each processor does a local sort then it finds the $P-1$ cutoff points that will separate the local data into $P$ segment. Then each processor sends those $P-1$ points to a central (reserved) processor. It will resort these $P(P-1)$ sample points and figure out the new $P-1$ cutoffs and sends them back to each processor, then each processor uses those as the bucket criteria. Thus

$$\begin{aligned} T_{sample}(n) &= T_{bucket}(n) + O\left(\tau \frac{n}{P} \log \frac{n}{P} + (\alpha + \beta P) + \tau P^2 \log P^2 + (\alpha + \beta P) \log P\right) \\ &= T_{bucket}(n) + O(P^2 \log P) \end{aligned}$$

In 2015 Spark has set new benchmark in sorting. It sorted 100TB of data in 23 minutes. http://spark.apache.org/news/spark-wins-daytona-gray-sort-100tb-benchmark.html

- Distributed Graph Operations

Distributed Level Synchronous BFS

Take advantage of distributed matrix multiplication: create adjacency (Boolean) matrix

$$\exists\ edge\ i \to j \iff a_{ij} = \text{True}$$

Recall we wrote `level_sync_BSF(G = (V,E), s)` function for parallel BFS. In it, there is a `processLevel(G, F[d], F[d+1], D)` function. We are going to replace this `processLevel` function.

Suppose there are $P$ processors and each stores a portion of the graph or actually a block of the adjacency matrix $A$. There is root (reserved) processor which will store the $D$ array that records the distance from each node of the graph to the starting node $s$.

The root processor will initialize $D$ and a frontier `F[0]` as array

```
D = [-1,-1,…,0,…,-1,-1]
F[0] = [0,0,…,1,…,0,0]
```

where the only different elements $-1, 1$ in `D` and `F[0]` are at the sth position, representing the start node. Then root processor will broad cast `F[0]` to all processors. And all processors will do distributed matrix multiplication

$$F[0] * A = F[1]$$

and send it back to the root processor. The root will use it to update $D$ array. If some element in $F[l]$ has already been visited, i.e. showing non -1 from $D$, it should be taken out of $F[l]$. Root processor will then the correct $F[l]$ to all processors and perform next matrix multiplication for $F[l+1]$.

Of course how matrix multiplication is performed depends on how the matrix was initial distributed.

<u>Graph Partitioning</u>

It is a very import problem for design of VLSI (very-large-scale integration) system.

Partition vertices set $V = V_0 \cup \ldots \cup V_{p-1}$ so that for any $i, j$
- $G_i \cap G_j = \emptyset$, this avoids double-count after combining results from each processor
- number of non-zero in $G_i \sim$ number of non-zero in $G_j$, balanced work. This is too hard. Graph partition problem only asks $|V_i| \sim |V_j|$
- minimize edges between $G_i, G_j$

Graph partition is NP-hard, meaning it's an upper bound of NP.

A problem is NP, stands for "non-deterministic polynomial time". It means we don't know if it can be solved in polynomial time, but if we are given a plausible answer to the problem, we can in fact verify the answer to see if it is correct or not in polynomial time. NP-complete means it is the lowest upper bound of NP and it is in NP, hence NP-complete = max(NP).

For NP-hard problem, we use heuristic algorithm.

We study three algorithms. All of them try to find a graph partition that cuts the graph into two equal vertices and minimize the cost$(G_1, G_2)$, which is the number of edges between $G_{1,2}$.

<u>Kernighan-Lin Algorithm</u>

Cut the graph into two arbitrary halves $V_1, V_2$. We want to pick two subsets $X_1 \subset V_1$, $X_2 \subset V_2$ with $|X_1| = |X_2|$, so that after swapping $X_1, X_2$, i.e.

$$V_1' = V_1 - X_1 + X_2, \qquad V_2' = V_2 - X_2 + X_1$$

Finding such $X_{1,2}$ is again NP-hard, so we should go heuristic.

Observation: consider two nodes $a \in V_1, b \in V_2$. Let $E[a] =$ number of edges $a$ connected to $V_2$, $I[a] =$ number of edges $a$ connected to $V_1$. Similarly define $E[b]$, $I[b]$. Let $c_{ab}$ be 1/0 if there is an edge between $a, b$. Such edge if exists is already counted in $E[a], E[b]$.

What is the change in cost if we swap $a, b$?

$$E[a] + E[b] - I[a] - I[b] - 2c_{ab}$$

That is external links become internal links.

KL-Algorithm:

- Pick arbitrary $V_1, V_2$
- for each node, compute $E[.], I[.]$. This part has run time

$$O(d)$$

where $d$ is the maximal degree of the graph.
- find a pair $a_1, b_1$ that maximizes

$$gain(a,b) = E[a] + E[b] - I[a] - I[b] - 2c_{ab}$$

- Record such pair, and remove them from graph (not actually remove them, but for the purpose of follow calculation)
- Recompute $E[.], I[.]$ for the rest of the nodes
- Again find a pair $a_2, b_2$ that maximizes

$$gain(a,b) = E[a] + E[b] - I[a] - I[b] - 2c_{ab}$$

- Continue as long as $gain(a,b) > 0$
- Swap these $(a_1, b_1), \dots, (a_k, b_k)$ from $V_1$ to $V_2$, we get $V'_1, V_2{}'$,
- Now repeat the whole process again for the new $V'_1, V_2{}'$, till all $gain(a,b) < 0$

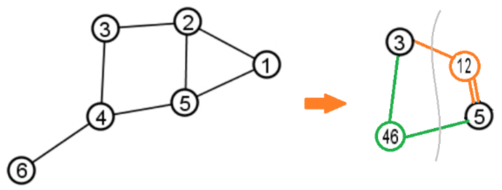
Since there are two loops ranging $O(|V|)$ for computing $gain$, total run time is

$$O(d|V|^2)$$

<u>Graph Coarsening</u>

The idea is to coarse a very large graph down to manageable size. When NP-hard is no longer hard, solve the graph partition problem for the small graph, then map the solution to the original graph.

How to coarse?



- start with a set of pairs of two nodes which have no edges among them. In the graph above. That will be $\{(1,2), (4,6)\}$. Such set is called maximal matchings
- combine pairs into one. Because node 5 goes to both nodes 1 and 2, the combined node (1,2) should have an edge with node 5 with weight 2.

- continue coarse using maximal matchings. Always try to combine edges with bigger weight first, like the node (1,2) and node 5, because once they combine, in the final graph they will not be separated. Such graph will likely close to be fully connected in the original graph, so they probably should not be separated.

How many coarsening stages in order to shrink the graph to final number of vertices = $k$? Suppose at each stage, the maximal matchings return the biggest possible set, i.e. half of the vertices, then in the best possible scenario,

$$\#\text{coarsening stages} = \Omega\left(\log\frac{|V|}{k}\right)$$

<u>Spectral Partition</u>

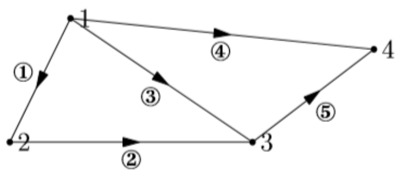
Define Laplacian matrix of a graph $L(G)$

- Off diagonals are negative of the adjacency matrix $(-A)$.
- Diagonal elements are the degrees of each node.

Note that: if it is a directed graph, treat it as undirected $(A \vee A^T)$. That is because graph partition doesn't care if it is directed or not. The cost function is the same, and for the spectral method, we will find eigenvalues and we will use the property of positive-semi-definite symmetric matrix. It is positive-semi-definite because assume $G$ is directed, (if not, convert it into a directed graph), then

$$L = C^T C$$

C = incidence matrix of a directed $G$. Columns represent nodes and rows represent edges. $-1$ indicates sink and $+1$ are the sources.



$$\text{incidence} = \begin{array}{cc} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} & \\ \begin{bmatrix} -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} & \begin{array}{c} ① \\ ② \\ ③ \\ ④ \\ ⑤ \end{array} \end{array}$$

Picture from http://ocw.mit.edu/courses/mathematics/18-06sc-linear-algebra-fall-2011/ax-b-and-the-four-subspaces/graphs-networks-incidence-matrices/MIT18_06SCF11_Ses1.12sum.pdf

Why is it called Laplacian? Consider the 2d mash network graph. What is its Laplacian matrix? It turns out its Laplacian is exactly the 2d Laplacian from pde.

Claim: Given a partition $V_{1,2}$, let $X$ be a column vector such that the $i^{\text{th}}$ element is $+1(-1)$ if the $i^{\text{th}}$ node is in $V_1(V_2)$, then the cost function

$$cost(V_1, V_2) = \frac{1}{4} X^T L(G) X = \#\, edges\ cutoff\ by\ the\ partition$$

Proof:

$$\sum_{i,j} X^T L X = \underbrace{\underbrace{\sum_{i=j} L_{ij} X_i X_j}_{\substack{sum\ of\ all\ degrees \\ =2\times total\ edges}} + \underbrace{\sum_{\substack{X_i X_j = 1 \\ i \neq j}} L_{ij} X_i X_j}_{-2\times internal\ edges}}_{2\times external\ edges} + \underbrace{\sum_{\substack{X_i X_j = -1 \\ i \neq j}} L_{ij} X_i X_j}_{2\times external\ edges}$$

QED.

The optimal problem is NP-hard.

$$\arg_X \min_{\substack{\sum X_i = 0 \\ X_i = \pm 1}} X^T L(G) X$$

So we relax it to,

$$\arg_X \min X^T L(G) X$$

Clearly by spectral theory, since all eigenvectors $\{v_k\}$ of $L$ are orthogonal, and we denote the associated eigenvalues $\{u_k\}$. Let

$$X = \sum \langle X, v_k \rangle v_k$$

The quadratic form

$$\min X^T L X = \min \sum u_k^2 \langle X, v_k \rangle^2$$

What are the eigenvectors of a typical Laplacian $-(\partial_x^2 + \partial_y^2)$? It is $\sin \sqrt{u_k} x + \cos \sqrt{u_k} y$. We want the smallest $u_k$. So it is the 1st eigen mode $v_1$, and it makes one intersection with the axis, i.e. half of a full wave period, which has exactly half positive and half negative values.

For general Laplacian of a graph, we take a heuristic solution

$$X = sign(v_1)$$

## 7. Memory/CPU Optimization

To achieve high performance, we should also optimize memories installed in the local machine. Real machine has hierarchy of memories: Disk, DRAM, L3, L2, L1/Cache, Registers. They are ranging from largest to smallest in memory size, and ranging from slowest to fastest.

For simplest, we first study two-level memory.

- Two-Level Memory Hierarchy

Von Neumann Model

- (local data rule) Processor only processes data on the fast memory, which is small in size, $Z$.
- (block transfer rule) Fast memory is connected to a slow memory. The transfer of data takes place in block size $L$ from the slow memory to fast memory.

Let's say the $x^{\text{th}}$ block is designated to addresses A[101] – A[120] and the $(x+1)^{\text{th}}$ block is designated to addresses A[121] – A[140]. If there is a variable array, $a$ of some size, such that part of array $a$ stored on the $x^{\text{th}}$ block and the rest of array $a$ stored on the $(x+1)^{\text{th}}$ block. To fetch $a$, both the $x^{\text{th}}$ block and the $(x+1)^{\text{th}}$ block will be transfer to the fast memory. Despite that size of $a$ is less than $L$, $2L$ of data will be sent to the fast memory.

Intensity, Balance, Time

Let $W$ be number of works to be processed by the processor, and $Q$ be the number of transfers that have to take place between fast and slow memory, then we define

$$Intensity = \frac{W(n)}{L \cdot Q(n, Z, L)}$$

Why is it defined? Let $\alpha$ be amortized time to move one unit of memory, $\tau$ be flop time. Then the total run time $T(n)$ is

$$\tau W \cdot \max\left(1, \frac{\alpha/\tau}{Intensity}\right) = \max(\tau W, \alpha L Q) \leq T(n) \leq \tau W + \alpha L Q = \tau W \left(1 + \frac{\alpha/\tau}{Intensity}\right)$$

The left bound assumes two processes can happen at the same time. $\alpha/\tau$ is a property purely controlled by the machine, call it

$$Balance = \frac{\alpha}{\tau}$$

Before our algorithms focused on $W$, now we see we have to pay attention to $Q(n, Z, L)$. Because machine balance improves frequently, for the algorithm to scale well, we would like to design algorithms or change fast memory ($Z$) so that

$$Intensity \gg Balance$$

- Two-Level Matrix Multiplication

Let's compare two methods. Conventional way: dot product of rows and columns. It has $W = O(n^3)$, because every entry is sum of $n$ products ($W = n + n - 1$), and there are $n^2$ of such entries. What about $Q$? Since matrix is stored by 1d raw (or 1d column), really

$$A[i, j] = A[(address\ of\ A[0]) + ni + j]$$

Thus to compute

$$C_{ij} = A_{il} B_{li}$$

Assume $L < n$, fast memory has to make, assume no cache! (see later section for cache)

$$Q = \left\lceil \frac{n}{L} \right\rceil + n = O(n)$$

So total

$$Q = O(n^3)$$

Clearly we moved move data than necessary.

Method Two: block multiply. Divide matrix into block of $b \times b$ size. We know $W = O(n^3)$ because consider a resulting block of $C$, it is the sum of $n/b$ blocks of $A$ times $n/b$ blocks of $B$. It takes works $W = b^3(n/b)$. There are total $(n/b)^2$ of such blocks. Thus $W = O(b^3(n/b)^3)$.

What about $Q$? To compute one block of $C$, we need sum of product of row blocks of $A$ and column blocks of $B$, it has to make $Q = \frac{b^2}{L}\frac{n}{b} = \frac{bn}{L}$ transfers, so total

$$Q = \frac{bn}{L}\left(\frac{n}{b}\right)^2 = O\left(\frac{n^3}{Lb}\right)$$

<u>Parallel Von Neumann Model</u>

Assume there are $P$ parallel processors, and they all access the fast memory at the same. Our previous discussions were still valid with a change of $W \to W/P$.

Besides worry about the existence of fast/slow memory, what else can we pay attention to?

<u>More physical aspects of algorithms to consider: Energy, Power, Time</u>

On a linux, one can program CPU frequency.
http://wiki.archlinux.org/index.php/CPU_frequency_scaling.
Clock freq $\propto V$, and <u>dynamic voltage scaling</u>

$$dynamic\ power = energy\ per\ gate\ (CV^2) * clock\ freq * activity\ factor$$
$$= \frac{energy\ consumed}{time}$$

Dynamic power is the addition energy/time pump into the system (for specific instruction to run), which on top the baseline energy which is constant as long as the system is turned on. Activity factor is number of cycles per switch.

Example: Let new frequency be

$$f' = \frac{f}{\sigma} \to new\ dynamic\ power = \frac{dynamic\ power}{\sigma^3}$$

If we don't do anything else, clearly the program will run slower and the total energy consumed will go down

$$T' = T\sigma, \ \ E' = \frac{E}{\sigma^2}$$

Now let us say we want to actually make the program run faster. Since the power is down by $\sigma^3$, we can use more processors without worrying overheat. So we increase parallelism.

$$P' = \sigma^3 P$$

By Brent's theorem

$$T'' \leq \left(D + \frac{W - D}{\sigma^3 P}\right)\sigma$$

The minimum is attained at

$$\sigma = \left(2\frac{W - D}{PD}\right)^{1/3}$$

- Two-Level Memory Sorting

We are now back to two-level memory model. We saw that typical data exchange is slower than computation. To increase Intensity, we would like to limit fast/slow data exchanges.

2-way Merge Sort

Sort $n$ elements. Assume the processor is sequential.

PHASE 1
- fetch $k$ blocks of size $L$, that $kL = Z$, from slow to fast memory, then sort. Overwrite them back to slow memory. Continue next $k$ blocks, till finish all $n$. End up having $n/Z$ sorted subsequences.

Total transfers

$$Q = O\left(\frac{n}{L}\right)$$

Total comparison computation

$$W = O\left(\frac{n}{Z} Z \log Z\right)$$

PHASE 2

To merge the $n/Z$ sorted subsequences, we use 2-way merge.

- Start with a pair of sorted subsequences, denoted $A_s, B_s$. Create an empty sequence $S_s$ in the slow memory as a place holder for the solution. Create $A_f, B_f, S_f$ of empty sequences with size $Z/3$ as place holders in the fast memory.
- Fetch $A_s, B_s$ to fill in $A_f, B_f$. Then do usual merge sort (compare elements by elements in the orders, and delete elements from $A_f, B_f$.), write solution to $S_f$. If $A_f, B_f$ become empty, fetch more from $A_s, B_s$. If $S_f$ becomes full, dump it to $S_s$.
- Repeat this for all pairs of subsequences and for any pairs of the resulting subsequences till $n$ is sorted

Transfers of one merge

$$Q = 4\frac{A_s}{L}$$

Comparison of one merge

$$W = A_s$$

Then the total

$$Q = \sum_{i=0}^{\log\frac{n}{Z}} 4\frac{Z2^i}{L}\frac{n}{Z2^{i+1}} = 2\frac{n}{L}\left(\log\frac{n}{Z} + 1\right), \quad W = \frac{Q}{4}L = \theta\left(n \log\frac{n}{Z}\right)$$

Combining two phases, we get

$$Q = O\left(\frac{n}{L}\log\frac{n}{Z}\right), \qquad W = O\left(n\log Z + n\log\frac{n}{Z}\right) = O(n\log n)$$

This turns out not to be the best we can do. Recall in the second phase, we fill the fast memory with three sequences $A_f, B_f, S_f$, but we only do very simple operations, which is a bad use of fast memory.

PHASE 2 ($k$-way)

To merge the $n/Z$ sorted subsequences, we use $k$-way merge.

- Start with $k = Z/L - 1$ sorted subsequences, denoted $A_s, B_s, \ldots$ Create an empty sequence $S_s$ in the slow memory as a place holder for the solution. Create $A_f, B_f, \ldots S_f$ of empty sequences with size $L$ as place holders in the fast memory.
- Fetch $A_s, B_s, \ldots$ to fill in $A_f, B_f, \ldots$. Then do usual $k$-merge sort. Compare the first elements of $A_f, B_f$,.. Find the extrema, write it to $S_f$, delete it, continue sorting. Notice if $k$ is large, don't want to compare $k$ elements, after deleting the extrema, adding one back in, then re-compare the whole list again, $O(k)$. Instead when $k$ is large, we reserve additional size $k$ space in fast memory. We build a min-heap for these $k$ element. Then deleting and adding become heapify operations, $O(\log k)$.
- The rest is the same. If $A_f, B_f, \ldots$ become empty, fetch more from $A_s, B_s, \ldots$. If $S_f$ becomes full, flush it to $S_s$.

Transfers of one $k$-way merge

$$Q = 2k\frac{A_s}{L}$$

Comparison (assume heap-based sort) of one $k$-way merge

$$W = k + kA_s \log k$$

Then the total

$$Q = \sum_{i=0}^{\log_k \frac{n}{L}} 2k\frac{A_s}{L}\frac{n}{kA_s} = \frac{2n}{L}\left(\log_k \frac{n}{L} + 1\right), \quad W = \frac{Q}{2} L \log k = O\left(n \log \frac{n}{L}\right)$$

Combining two phases, we get

$$Q = O\left(\frac{n}{L}\log_{\frac{Z}{L}} \frac{n}{L}\right), \qquad W = O\left(n \log Z + n \log \frac{n}{L}\right) = O(n \log n)$$

This turns out to be the best we can do.

<u>The Lower bound of Sort Transfers</u>

Claim: The lower bound of comparison based sort on external memory is

$$Q(n, Z, L) = \Omega\left(\frac{n}{L}\log_{Z/L} \frac{n}{L}\right)$$

Proof:

For $n$ distinct items, there are $n!$ orderings. Only 1 of them gives what we want. After we have seen $L$ items out of the $n$ items, and sort these $L$ items. How many orderings out of initial $n!$ orderings are still possible?

The answer $= n!/L!$. Why? Consider the rest $n - L$ unseen items. They are still in random. For each of these correct sequences, if we were allowed to let these seen items to be in random order again, we would get

$$\text{answer} \times L! = n! \rightarrow \text{answer} \ = \frac{n!}{L!}$$

Let

$$K(t) = \#\text{ of correct orderings after the t}^{\text{th}}\text{ transfer}$$

Consider $K(t)$, assume the fast memory was full. Now flush out $L$ data and add additional new $L$ data to $Z$, and we can sort them, therefore the randomness from $K(t-1)$ is reduced to

$$K(t) = \frac{K(t-1)}{\binom{Z}{L} L!}$$

The $\binom{Z}{L}$ factor gives the number of ways to find seats in $Z$ and insert $L$ data into these seats, and $L!$ Gives the number of ways to order $L$. Now these freedoms are gone.

So the asymptotic formula becomes

$$K(t) = \frac{n!}{\left[\binom{Z}{L} L!\right]^t}$$

We want to equal $K(t) = 1$ and solve for $t$. Applying Stirling's

$$1 = \frac{n!}{\left[\binom{Z}{L} L!\right]^t} \rightarrow 0 = n \ln n - t[Z \log Z - (Z-L)\log(Z-L)]$$

We get

$$t \approx \frac{n \ln n}{Z \ln Z - (Z-L)\log(Z-L)} \geq \frac{n(\ln n - \ln L)}{L(\ln Z - \ln L)} = \frac{n}{L} \log_{Z/L} \frac{n}{L}$$

QED

- Two-Level Memory Searching

<u>The Lower bound of Search of a Sorted Sequence Transfers</u>

A simple binary search will fetch $L$ data from a list in the middle, then decide to move left or right, in worst case total transfers

$$Q = O\left(\log \frac{n}{L}\right)$$

Can we do better? Yes, but we have to build store the list into a specific layout: B-tree.

Recall a B-tree whose nodes have at most $B$ keys can have $(B+1)$ children. E.g. Well-known (2,3) tree is a B-tree with $B = 2$. The ordering of a B-tree is very similar to a binary tree, however inserting is very different. Generally speaking, inserting into a binary tree will create a leaf down; while inserting into a B-tree will try to bubble it up till it finds an upper parent that is not full. All efforts are trying to regulate the tree height to $\log_{B+1} n$. In term of our searching using fast-slow memories, let $B = \theta(L)$. In worst case total transfers

$$Q = O\left(\frac{\log n}{\log L}\right)$$

That is a lot better than $\log \frac{n}{L} = \log n - \log L$.

Claim: The lower bound of search on sorted sequence is

$$Q(n, Z, L) = \Omega(\log_L n)$$

Proof:

Consider the sorted list as values of a random variable $X$. Borrow ideas from information theory, if $X$ can take $n$ different values, then we need

$$\log n$$

number of bits to represent it. If we are clever, we can pick the optimal $L$ values to test $X$, so that $\log L$ out of the $\log n$ bits can be eliminated. If every read can eliminate $\log L$, then total number of transfers is

$$Q = O(Log_L n)$$

QED

- Cache and Paging

We assume machine automatic organizes fast memory (Cache) by these rules

- Fast memory is divided into blocks of size $L$, $Z = \Omega(L^2)$ is because of matrix multiplication.
- When the program calls LOAD, to write some variable value to the register, the processor will first check the cache if it is there. If it is not (called cache miss), a copy will be stored in cache.

- Similarly when the program calls STORE, to write register value to the slow memory, the processor will first check the cache. If is not (cache miss), a copy will be stored in cache.
- The cache is fully associative, meaning a continuous blocks of data in the slow memory doesn't have to be store continuous blocks in the cache.
- When cache is full, it will evict the block that is least recently used (LRU).

So the total number of transfer is determined by

$$Q = \text{cache miss}$$

Is the least recently used (LRU) eviction policy good? It is very good. It is as asymmetrically good as the best it can be, which is the optimal eviction, which is to evict items that will be used least in the future.

Lemma (competitiveness):

$$Q_{LRU}(n, Z, L) \leq 2 \cdot Q_{OPT}\left(n, \frac{Z}{2}, L\right)$$

Proof:
Consider a very long instruction segment of a program. The piece of instruction contains $Z$ unique addresses. The maximum number of transfer to clear out the cache and to run the piece of instruction by LRU will be

$$Q_{LRU}(Z) \leq Z$$

While for OPT cache, because let us assume that it is so efficient that before it hits the piece of instruction, its cache contains exact the first $Z/2$ of the addresses in the instruction, hence it can go the first half of the instruction without evicting anything, but after that, to finish the instruction, it has to evict $Z/2$ of addresses, so

$$Q_{OPT}\left(\frac{Z}{2}\right) \geq \frac{Z}{2}$$

QED

Corollary: if (regularity condition)

$$Q_{OPT}\left(n, \frac{Z}{2}, L\right) = O\big(Q_{OPT}(n, Z, L)\big)$$

then

$$Q_{LRU}(n, Z, L) = \theta\big(Q_{OPT}(n, Z, L)\big)$$

Proof:

$$Q_{OPT}(Z) \leq Q_{LRU}(Z) \leq 2Q_{OPT}\left(\frac{Z}{2}\right) = O\big(Q_{OPT}(Z)\big)$$

QED

- Cache Oblivious Algorithms

Write algorithms that let automatic cache do the work still achieve the optimal result as if we specified how to read/write between fast/slow memories.

Cache Matrix Multiplication

Recall cache aware block multiplication of block size $b \times b$. Assume $b^2 = Z$

$$Q = O\left(\frac{n^3}{L\sqrt{Z}}\right)$$

Now we write cache oblivious code that will match the same transfers.

```
mm(n, A, B, C)  //n size of matrix
      //divide and conquer
      if n <- 1 then  C <- C + A*B
      else
              for i <- 1 to 2
                       for j <- 1 to 2
                                 for k <- 1 to 2
                                            mm(n/2, A_ik, B_kj,  C_ij)
                           //e.g. A_11 means upper left block, etc
```

Consider a small $n_b$ such that $3n_b^2 \leq Z$, so the program will cache everything at the beginning. For $n > n_b$, when it divides and conquers into 8 smaller function calls and

$$Q(n, Z, L) = \begin{cases} \theta\left(\frac{n^2}{L}\right) & n < n_b \\ 8 \cdot Q\left(\frac{n}{2}, Z, L\right) + O(1) & n > n_b \end{cases}$$

By the master theorem

$$Q(n, Z, L) = O\left(\frac{n^3}{L\sqrt{Z}}\right)$$

Cache Binary Search

Recall we showed B-tree attained the minimum.

$$Q = O(\log_L n)$$

Now we show another layout: Van Emde Boas tree. The idea is similar to the matrix block multiply above. Divide a binary tree into equal three trees; one on top, and two on the bottom. Store those three subtrees as min-heap. Continue divide subtree into three subsubtrees, do this recursively. At some point, the subtree size will become $L$, the height of the tree of the subtree is $\log n / \log L$. Why? That is because the original tree height is $\log n$, and each subtree's height is $\log L$, so the height of the tree of the subtree is the ratio.

# The Future: Quantum Computing

## 8. Quantum Information & Entanglement

### • Mix States

qubit = quantum + bit, which is superposition of

$$|\psi\rangle = a|1\rangle + b|0\rangle, a, b \in \mathbb{C}, |a|^2 + |b|^2 = 1$$

where $|a|^2$ = probability of $|\psi\rangle$ in state $|1\rangle$.

Combine two systems, create joint (pure) state (soon we will talk about mix states)

$$|\psi_1, \psi_2\rangle = |\psi_1\rangle \otimes |\psi_2\rangle = a_1a_2|1,1\rangle + a_1b_2|1,0\rangle + b_1a_2|0,1\rangle + b_1b_2|0,0\rangle$$

One can create projection operators or called the density matrix

$$P = |\psi_1\rangle\langle\psi_1|\ apply\ to\ (|\psi_2\rangle) \rightarrow (a_1a_2 + b_1b_2)|\psi_1\rangle$$

and

$$|a_1a_2 + b_1b_2|^2 = \text{the expectation value of } |\psi_2\rangle\ applied\ to\ P\ or\ project\ to\ |\psi_1\rangle$$

What does $P = |\psi_1\rangle\langle\psi_1|$ look like in matrix form?

$$\begin{pmatrix} a \\ b \end{pmatrix} \begin{pmatrix} a^* & b^* \end{pmatrix} = \begin{pmatrix} |a|^2 & ab^* \\ ba^* & |b|^2 \end{pmatrix}$$

What is $P^2$?

$$P^2 = P$$

Notice the trace of $P^2$ is 1. Similarly, if we do this for $|\psi_1, \psi_2\rangle$ above, we get trace 1 too.

The trace of an operator is thus defined to be

$$Tr(O) = \sum_{\substack{n \in orthogonal \\ states\ spans\ Hilbert}} \langle n|O|n\rangle$$

Trace is nice because it is invariant of bases. Also it is cyclic $Tr(ABC) = Tr(BCA)$.

The density matrix is nice too. If we want to find expectation of any operation $O$ acts on $|\psi\rangle$, we can use its density matrix

$$\langle\psi|O|\psi\rangle = Tr(\langle\psi|O|\psi\rangle) = Tr(O|\psi\rangle\langle\psi|) = Tr\left(OP_\psi\right).$$

This suggests we can use solely the density matrix to represent the state.

What if we create a Frankenstein state?

$$P_{Frank} = \sum_{\substack{n \in some\ (not\ necessarily) \\ orthogonal\ states}} \lambda_k\, |\psi_k\rangle\langle\psi_k|$$

with all $\lambda_k \in [0,1]$, and $\sum\lambda_k = 1$.

Clearly

$$Tr(P_{Frank}) = \sum\lambda_k = 1$$

That is why we need $\sum\lambda_k = 1$. Since $P_{Frank}$ is made of not necessarily orthogonal states, in general

$$\langle\psi_k|P_{Frank}|\psi_k\rangle \neq \lambda_k$$

But what about $P^2 = P$?

$$P_{Frank}^2 = \sum\lambda_k^2|\psi_k\rangle\langle\psi_k|$$

and

$$Tr(P_{Frank}^2) = \sum\lambda_k^2 \leq 1$$

Ha-La, the only way $P^2 = P$ is when all except one $\lambda_k = 1$. So when $P^2 \neq P$ equivalently when $Tr(P^2) \neq 1$, we say

$$P_{Frank} = a\ mix\ state$$

Because trace is invariant, there is no way a pure state $|\psi\rangle$ can give arise to $P_{Frank}$ by doing $|\psi\rangle\langle\psi|$.

- Entropy and Information

We define entropy of information of the state

$$S(P) = -Tr(P \log_2 P)$$

To log matrix, we want to first diagonalize $P$. This can always be achieved because $P$ is Hermitian.

$$S(P) = -\sum_{e=1}^{|H|} (\lambda_e \log_2 \lambda_e)$$

where $\lambda_e$ are eigenvalues. Notice if $\lambda_e = 1 \; or \; 0$, $\lambda_e \log_2 \lambda_e = 0$. Thus

$$S(Pure) = 0, \qquad S(Mix) > 0$$

- Entanglement

An example of mix state

$$P = \frac{1}{2}(|0\rangle\langle 0| + |1\rangle\langle 1|)$$

To create it, we start from a pure joint state,

$$|\psi_{AB}\rangle = \frac{|00\rangle_{AB} + |11\rangle_{AB}}{\sqrt{2}}$$

Then take partial trace of the second particle

$$\sum_{\substack{states\ of \\ second\ particle \\ n=0,1}} \left[\langle n| \left(\frac{|00\rangle + |11\rangle}{\sqrt{2}}\right)\left(\frac{|00\rangle + |11\rangle}{\sqrt{2}}\right)^* |n\rangle\right] = \frac{1}{2}(|0\rangle\langle 0| + |1\rangle\langle 1|)$$

Now we see 1st particle is shewed. There is no way it can be pure again. We say

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} = an\ entangled\ state$$

Clearly if $|\psi_{AB}\rangle$ were some simple product

$$|\psi_{AB}\rangle = |\Phi_A\rangle|\phi_B\rangle$$

Then partial trace will return pure state.

Another example, check

$$|\psi_{AB}\rangle = \frac{|00\rangle + |10\rangle + |01\rangle + |11\rangle}{2} = not\ entangled\ or\ separable$$

We define a measure of the degree of entanglement as the entropy of after taking partial trace with respects to either $A$ or $B$,

$$S\left(Tr_{A\ or\ B}\left(P_{|\psi_{AB}\rangle}\right)\right)$$

Claim: either way the value is the same.

Proof: start with a pure joint state, (we need a pure state, because we need each $|\Phi_A\rangle, |\phi_B\rangle$ orthogonal in their own $H$ spaces. Otherwise the claim is not true, so quantifying mixed entanglement states remain a hard problem)

$$|\psi_{AB}\rangle = \sum_k \rho_k\ |\Phi_A\rangle|\phi_B\rangle$$

Then

$$Tr_A\left(P_{|\psi_{AB}\rangle}\right) = \sum_k \rho_k^2\ |\Phi_A\rangle\langle\Phi_A| \rightarrow S\left(Tr_A\left(P_{|\psi_{AB}\rangle}\right)\right) = \sum_k \rho_k^2 \log \rho_k^2 = Tr_B\left(P_{|\psi_{AB}\rangle}\right)$$

QED

Suppose we measure 2nd particle and find it at $|1\rangle$, that is the state collapsed

$$_B\langle 1|\left(\frac{|00\rangle_{AB} + |11\rangle_{AB}}{\sqrt{2}}\right) \sim |1\rangle_A$$

For entangled state, by measuring one particle, we change the state of the other. For un-entangled

$$_B\langle 1|\left(\frac{|00\rangle + |10\rangle + |01\rangle + |11\rangle}{2}\right) \rightarrow |1\rangle_A + |0\rangle_A$$

State of the other particle doesn't change.

## 9. From Laser to Quantum Computer

One promising way to build quantum computer is to use laser and single atom.

- Coherent Mixture States & Lasers

Coherent state $|A\rangle$ exhibits classical behavior (see my quantum mechanics note I section 2.2). It is the eigenstate of the lowing operator $a$,

$$a|A\rangle = A|A\rangle$$

Since $a$ is not Hermitian, $A$ is complex, one can express $|A\rangle$ in terms of energy states

$$|A\rangle = e^{-\frac{|A|^2}{2}} \sum_n \frac{A^n}{\sqrt{n!}} |n\rangle = e^{Aa^+}|0\rangle e^{-\frac{|A|^2}{2}}$$

How does $|A\rangle$ evolve?

$$e^{-i\frac{Ht}{\hbar}}|A\rangle = e^{-i\frac{Ht}{\hbar}} e^{Aa^+} e^{i\frac{Ht}{\hbar}} e^{-i\frac{Ht}{\hbar}} |0\rangle e^{-\frac{|A|^2}{2}} = e^{Aa^+} e^{-iwt}|0\rangle e^{-\frac{i}{2}wt} e^{-\frac{|A|^2}{2}} = e^{-iwt}|A\rangle e^{-\frac{i}{2}wt}$$

Similarly

$$ae^{-i\frac{Ht}{\hbar}}|A\rangle = Ae^{-iwt}e^{-i\frac{Ht}{\hbar}}|A\rangle$$

Combining the two gives

$$|A(t)\rangle \rightarrow e^{-\frac{i}{2}wt}\left|A(0)e^{-iwt}\right\rangle$$

To measure this evolution of phase $e^{-\frac{i}{2}wt}$, one can use beam splitter and ask split beams to travel different optical paths and then recombine them to see interference.

Why have coherent states to do with beams? Recall quantization of EM, (for deviation see my quantum mechanics II notes chapter 4)

$$\vec{E} = i \sum_{\lambda,\vec{k}} \sqrt{\frac{2\pi\hbar w}{V}} \left[\vec{\epsilon}_\lambda a e^{i\vec{k}\cdot\vec{r}} - a^+ \vec{\epsilon}_\lambda^{\,*} e^{-i\vec{k}\cdot\vec{r}}\right]$$

where $V$ = cubic space in which EM is in, $w = c\left|\vec{k}\right|$, $\lambda = 1{,}0$, $\vec{\epsilon}_\lambda$ = two polarizations. $\vec{k}$ is the wave vector. $\vec{r}$ = position vector. $a$ = annihilation operator, we use convention $\epsilon_0 = 1/4\pi$. The moral of the story is that lights are photons. Furthermore if we make a (maximum because we have no knowledge of the phases) mixture of all coherent states of same amplitude, i.e. $|A\rangle = |A|e^{i\phi}$, same $|A|$, allow $\phi$ to vary

$$P = \frac{1}{2\pi}\int_0^{2\pi} d\phi\, |A\rangle\langle A| = e^{-|A|^2} \sum_n \frac{A^{2n}}{n!} |n\rangle\langle n|$$

We arrive the most logical answer, a Poisson distribution.

- Quantum Transistor and Gates

All quantum evolutions are given by unitary operators.

$$i\hbar\frac{d|\psi\rangle}{dt} = H|\psi\rangle \to |\psi\rangle_{t_2} = U(t_1, t_2)|\psi\rangle_{t_1}$$

This restricts the possible actions one can do. In particular,

No-cloning / No deleting Theorem: Given two bits $|x\rangle|y\rangle$, one cannot make a copy of $x$ to $y$.

Proof: Suppose there exists such $U$

$$U(|x\rangle|y\rangle) = |x\rangle|x\rangle$$

Try

$$U(|0\rangle|1\rangle) = |0\rangle|0\rangle \;\; and\; U(|0\rangle|0\rangle) = |0\rangle|0\rangle$$

then take dot of the two equations, get

$$0 = 1$$

Contradiction. QED

<u>Quantum Pauli-X (Not) Gate</u>

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Then

$$X|0\rangle = |1\rangle \;\; and\; X|1\rangle = |0\rangle$$

<u>Quantum Hadamard Gate</u>

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Then

$$H|0\rangle = \frac{|1\rangle + |0\rangle}{\sqrt{2}} \; and\; H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

<u>Quantum CNot (control not) Gate</u>

$$CNOT = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{pmatrix}$$

Then

$$U\begin{pmatrix} |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{pmatrix} = \begin{pmatrix} |00\rangle \\ |01\rangle \\ |11\rangle \\ |10\rangle \end{pmatrix}$$

Hence the first bit is an enabler, when it is on, the second bit gets negated.

Example.

To create the entangled state

$$\frac{|11\rangle + |00\rangle}{\sqrt{2}}$$

Create two $|0\rangle, |0\rangle$ particles, send the first one to $H$, then combine the second one. The send the to CNOT

$$|0\rangle|0\rangle \rightarrow \frac{|1\rangle|0\rangle + |0\rangle|0\rangle}{\sqrt{2}} \rightarrow \frac{|11\rangle + |00\rangle}{\sqrt{2}}$$

If we send to CNOT again, we detangle them

$$\frac{|11\rangle + |00\rangle}{\sqrt{2}} \rightarrow \frac{|10\rangle + |00\rangle}{\sqrt{2}}$$

## 10. Quantum Algorithms

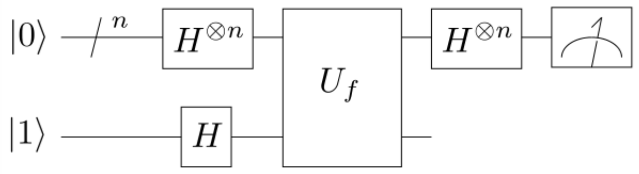
- ### Deutsch's Problem

$$f: \{0,1\}^n \rightarrow \{0,1\}$$

Question: is $f$ a constant or a balance function? We are told it is one of the two choices. A balance function output half 1 and half 0.

$f$ is called an oracle, a black box. We cannot break in the internal mechanism of $f$. We can only inquire $f$.

Classically need to do $2^{n-1}+1$ evaluations. Quantum computer does 1 evaluation of $f$.



Picture from https://en.wikipedia.org/wiki/Deutsch%E2%80%93Jozsa_algorithm

- prepare $n+1$ qbits. The first $n$ are $|0\rangle$, the last one is $|1\rangle$. Send them to Hadamard

$$\frac{|1\rangle+|0\rangle}{\sqrt{2}}\frac{|1\rangle+|0\rangle}{\sqrt{2}}\dots\frac{|0\rangle-|1\rangle}{\sqrt{2}}=\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}}$$

- apply $f$ to the $x$ particles, and add it to the last.

$$U_f|x\rangle|y\rangle=|x\rangle|f(x)+y\rangle$$

That is CNot, because + 0 does nothing, but +1 is flip.

$$\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x\rangle\frac{|f(x)\rangle-|f(x)+1\rangle}{\sqrt{2}}=\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x\rangle\,(-1)^{f(x)}\frac{|0\rangle-|1\rangle}{\sqrt{2}}$$

- trace out the last bit, apply Hadamard

$$\frac{1}{2^n}\sum_{y\in\{0,1\}^n}\sum_{x\in\{0,1\}^n}|y\rangle\,(-1)^{x\cdot y+f(x)}$$

Then measure $y=\{0\}^n$

$$\left|\frac{1}{2^n}\sum_{x\in\{0,1\}^n}|0\dots0\rangle\,(-1)^{f(x)}\right|=1\ if\ f\ is\ contant, 0\ otherwise$$

- Grover Algorithm

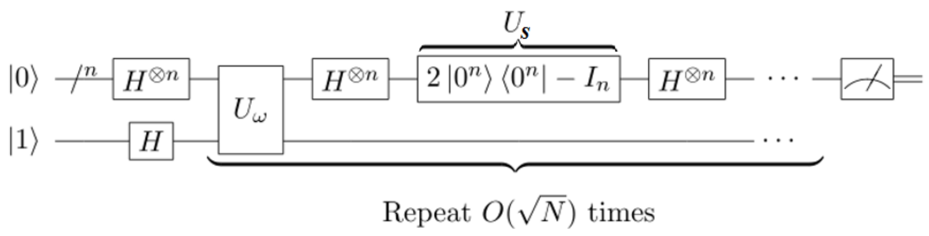
Problem: Give a function

$$f:\{1,\dots,N\}\to\{0,1\}$$

and we know there is only one $x^* \in \{1, \dots, N\}$ such that $f(x^*) = 1$, otherwise 0. Our task is to find $x^*$.

Classical method will do $O(N)$ searches. Quantum computer will do

$$O(\sqrt{N})$$



Picture from https://en.wikipedia.org/wiki/Grover%27s_algorithm

- apply Hadamard

$$\frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

- then apply $U_w|x\rangle = (-1)^{f(x)}|x\rangle$

$$\frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{f(x)}|x\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2^n}} \left( \sum_{x \in \{0,1\}^n} |x\rangle - 2|x^*\rangle \right) \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

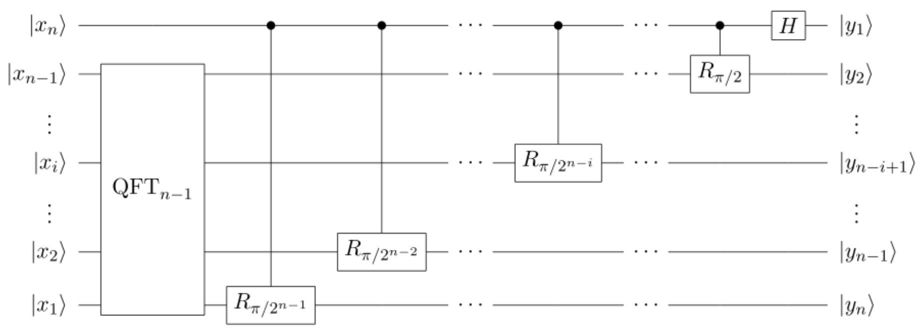
- Apply next three operators

$$\left[ \frac{2^n - 4}{2^n} \sum_{x \in \{0,1\}^n} |x\rangle + \frac{2}{2^n} |x^*\rangle \right] \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

- Then we continue the loop.

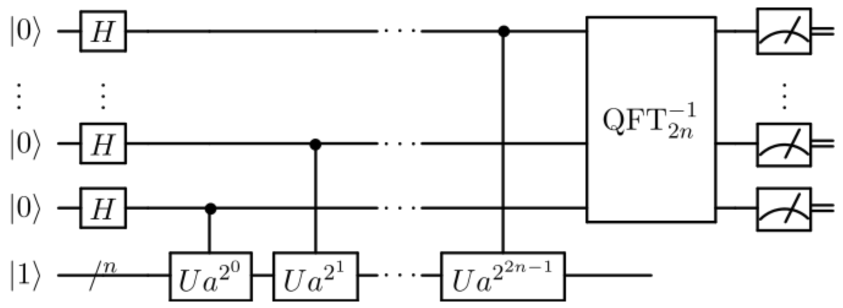
After each loop the amplitudes of other states vs $|x^*\rangle$ is reduced by $1/\sqrt{2}$. After $\sqrt{N}$ steps, the probability of getting $|x^*\rangle \geq 1/2$.

- Quantum Fourier Transform



Picture from https://en.wikipedia.org/wiki/Quantum_Fourier_transform

- Shor Algorithm (factorization)



Picture from https://en.wikipedia.org/wiki/Shor%27s_algorithm

# Reference

## 11. Books

- Gilbert Strang, *Computational Science and Engineering*, (with M code) Wellesley-Cambridge 2007, MIT
- David Patterson, Computer Organization and Design, 5th ed Berkeley
- Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, MIT
- Terry Tao, *Topics in Random Matrix Theory*, AMS 2012 UCLA
- Peter Lax, *Linear Algebra and Its Applications*, Wiley-Interscience, NYU 2007
- Cleve Moler, *Numerical Computing with MATLAB*, (with M code), MATLAB 2004
- Lloyd Trefethen, *Spectral Methods in MATLAB* (with M code), SIAM 2001
- Jose Unpingco, *Python for Signal Processing* (with Py code), Springer, 2014

## 12. Courses

- Jim Demmel, *Applications of Parallel Computers* (slides, notes, videos), Berkeley
- Leslie Greengard, *A Short Course on Fast Multipole Methods* (Notes), NYU

- Erik Demaine, *Advanced Data Structures*, MIT
- Mark Tygert, *Descendants of Fast Multipole Methods and Caldero'n-Zygmund Theory* (Notes), NYU
- Sinan Gunturk, *Wavelets, Approximation Theory, and Signal Processing* (note by Even Chou), NYU
- Richard Gonsalves, *Computational Physics I*, *Computational Physics II*, Buffalo
- Kristjan Haule, *Physics Applications of Computers*, Rutgers University
- Richard Furnstahl, *Computational Physics*, Ohio State University
- John Preskill, *Quantum Computation*, Caltech
- Mark Oskin, *Quantum Computing - Lecture Notes*, University of Washington
- Steven van Enk, *Mixed States and Pure States* (notes), University of Oregon
- Ryan O'Donnell, John Wright, *Quantum Computation and Information*, Carnegie Mellon University
- Steven Johnson, *Introduction to Numerical Methods* (with Py code), MIT
- Alan Edelman, *Numerical Computing with Julia* (with Julia code), MIT
- David Evans, *Operating Systems* (Rust), University of Virginia

## 13. On-line Course

- Rich Vuduc, *High Performance Computing*, Udacity, George Tech
- David Luebke, *Intro to Parallel Programming* (CUDA), Udacity, NVIDIA
- Milos Prvulovic, *High Performance Computer Architecture*, Udacity, George Tech
- Martin Odersky, *Functional Program Design in Scala*, Coursera, École Poly
- Heather Miller, *Big Data Analysis with Scala and Spark*, Coursera, École Poly
- Srinivas Devadas, Erik Demaine, *Design and Analysis of Algorithms,* MIT OCW
- Gilbert Strang, *Mathematical Methods for Engineers* (I, II) MIT OCW
- Wolfgang Ketterle, *Atomic and Optical Physics* (I, II) MIT OCW
- Hitoshi Murayama, *From the Big Bang to Dark Energy*, Coursera, University of Tokyo, Berkeley

## 14. Computing

- OpenMP https://computing.llnl.gov/tutorials/openMP/
- MPI library https://computing.llnl.gov/tutorials/mpi/
- CUDA https://developer.nvidia.com/cuda-zone
- MapReduce http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
- Spark http://spark.apache.org/
- Big Data Finance http://bigdatafinanceconference.com (NYU)
- http://quantlib.org/index.shtml
- https://cloud.sagemath.com/
- http://aws.amazon.com/
- http://julialang.org/