Computational Requirements of Scientific Applications
Computational Science Applications

Simulation of physical phenomena

- fluid flow over aircraft (Boeing 777 designed by simulation)
- fatigue fracture in aircraft bodies
- bone growth
- evolution of galaxies

Two main approaches

- **continuous methods**: fields and partial differential equations (pde’s) (eg. Navier-Stokes equations, Maxwell’s equations, elasticity equations..)
- **discrete methods**: particles and forces between them (eg. Gravitational/Coulomb forces)

We will focus on pde’s in this lecture.
Modeling physical phenomena using pde’s

PDE: \( L \ u = f \)

eg: \( \left( \frac{\delta^2}{\delta x^2} + \frac{\delta^2}{\delta y^2} \right) u = 0 \)

Domain: \( \Omega \)

Boundary conditions: on \( \delta \Omega \)

\[ u(x,y) = x + y \mid (x,y) \text{ on } \delta \Omega \]

General technique: find an approximate solution that is a linear combination of known functions

\[ u^* (x,y) = \sum_i c_i \Phi_i (x,y) \]

Question: How do we choose the known functions?
How do we find the best choice of c’s, given the functions?
Choice of known functions:

- periodic boundary conditions: can use sines and cosines
- finite element method: generate a mesh that discretizes the domain
  use low degree piecewise polynomials on mesh

1-D example

2-D example

Mesh generation
Finding the best choices of the coefficients:

Analogy with Fourier series:

\[ f(x) = a_0 + \sum_i a_i \cos(ix) + \sum_i b_i \sin(ix) \]

How do you find ‘best’ choices for a’s and b’s?

\[
\begin{align*}
\int_{-\pi}^{+\pi} f(x) \cos(kx) \, dx &= \int_{-\pi}^{+\pi} \left( a_0 + \sum_i a_i \cos(ix) + \sum_i b_i \sin(ix) \right) \cos(kx) \, dx \\
&= \int_{-\pi}^{+\pi} a_k \cos(kx) \cos(kx) \, dx \\
&= a_k \pi
\end{align*}
\]

**Key idea:**
- residual  \( f(x) - a_0 + \sum_i a_i \cos(ix) + \sum_i b_i \sin(ix) \)
- weight residual by known function and integrate to find corresponding coefficient
Weighted Residual Technique:

Residual: \((L \ u^* - f) = (L \ (\sum_{i}^{N} c_i \phi_i) - f)\)

Weighted Residual: \((L \ (\sum_{i}^{N} c_i \phi_i) - f) \ \phi_k\)

Equation for \(k^{th}\) unknown: \(\int_{\Omega} \phi_k * (L \ (\sum_{i}^{N} c_i \phi_i) - f) \ dV = 0 \ \implies \)

If the differential equation is linear:

\[ \sum_{k=1}^{N} c_k \int_{\Omega} \phi_k^* \ L \ \phi_1 \ dV + \ldots + c_N \int_{\Omega} \phi_k^* \ L \ \phi_N \ dV = \int_{\Omega} \phi_k^* \ f \ dV \]

This system can be written as

\[ K \ c = b \]

where

\[ K(i,j) = \int_{\Omega} \phi_i^* \ L \ \phi_j \ dV \]

\[ b(i) = \int_{\Omega} \phi_i^* \ f \ dV \]

Key insight: Calculus problem of solving PDE is converted to linear algebra problem of solving \(K \ c = b\) where \(K\) is sparse
Solving system of linear algebraic equations:

- $Kc = b$

Orders of magnitude for realistic problems:
- large ($\sim 10$ million unknowns) (roughly equal to number of mesh points)
- sparse ($\sim 100$ non-zero entries per row) (roughly equal to connectivity of a point)
- same $K$, many $b$'s in some problems

Algorithms:
- iterative methods (Jacobi, conjugate gradient, GMRES)
  
  start with an initial approximation to solution and keep refining it till you get close enough
- factorization methods (LU, Cholesky, QR)
  
  factorize $K$ into LU where $L$ is lower triangular and $U$ is upper triangular
  
  $LUc = b$
  
  Solve for $c$ by solving two triangular systems
Jacobi: a (slow) iterative solver

Example:

\[ 4x + 2y = 8 \]
\[ 3x + 4y = 11 \]

Iterative system:

\[
x_{n+1} = \frac{8 - 2y_n}{4}
\]
\[
y_{n+1} = \frac{11 - 3x_n}{4}
\]

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>2</td>
<td>0.625</td>
<td>1.375</td>
<td>0.8594</td>
<td>1.1406</td>
<td>0.9473</td>
<td>1.0527</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>2.75</td>
<td>1.250</td>
<td>2.281</td>
<td>1.7188</td>
<td>2.1055</td>
<td>1.8945</td>
<td>2.0396</td>
</tr>
</tbody>
</table>
Matrix view of Jacobi Iteration

Iterative method for solving linear systems $Ax = b$

Jacobi method: $M * X_{k+1} = (M - A) * X_k + b$ (M is DIAGONAL(A))

while (not converged) do
  do $k = 1..N$
    $Y[k] = 0.0$
    Initialization
  do $j = 1..N$
    do $i = 1..N$
      $Y[i] = Y[i] + A[i,j]*X[j]$  
      Matrix-vector product
  do $i = 1..N$
    $X[i] = (b[i] - Y[i])/A[i,i] + X[i]$  
    SAXPY operations
check convergence

Matrix-vector product: $O(N^2)$ work
SAXPY, Inner product: $O(N)$ work
Most of the time is spent in matrix-vector product.
Lesson for software systems people: optimize MVM
Reality check:

• Jacobi is a very old method of solving linear systems iteratively.

• More modern methods: conjugate gradient (CG), GMRES, etc. converge faster in most cases.

• However, the structure of these algorithms is similar: MVM is the key operation.

• Major area of research in numerical analysis: speeding up iterative algorithms further by\textit{ preconditioning}.
Tangential Discussion

- Calculus problem $Lu = f \Rightarrow$ linear algebra problem $Kc = b$.
- In some problems, we need to solve for multiple variables at each mesh point (temperature, pressure, velocity etc.) ⇒ solve many linear equations with same $K$, different $b$’s.
- This is viewed as matrix equation $KC = B$ where $C$ and $B$ are matrices.
- Algorithms for solving single system can be used to solve multiple systems as well.
- Key computation in iterative methods: matrix-matrix multiplication (MMM) rather than matrix-vector multiplication (MVM).
- Non-linear pde’s lead to non-linear algebraic systems which are solved iteratively (Newton’s method etc.).
  Key computation: MMM or MVM.
Computational Requirements

Let us estimate storage and time requirements.

- Assume $10^6$ mesh points (rows/columns of $A$)
- Assume iterative solver needs 100 iterations to converge
- Assume simulation runs for 1000 time steps.

One MVM requires roughly $10^{12}$ flops

$\Rightarrow$

Overall simulation requires $10^{17}$ flops and $10^{12}$ bytes of storage!

Can we do better?
1-D case

\[ K(i,j) = \int_{\Omega} \phi_i * L(\phi_j) \, d\Omega \]

Structure of the K matrix for any pde: \( K[i,j] = 0 \) if \( \phi_i \) and \( \phi_j \) do not overlap!

For our example, K is

\[
\begin{bmatrix}
  x & x & 0 & 0 & 0 \\
  x & x & x & 0 & 0 \\
  0 & x & x & x & 0 \\
  0 & 0 & x & x & x \\
  0 & 0 & 0 & x & x
\end{bmatrix}
\]

Half the entries are zero!

In 2-D and 3-D, an even larger percentage of matrix is zero!

Typical 3-D numbers: \( 10^6 \) rows but only 100-500 non-zeros per row!

Matrix is sparse.
Exploiting sparsity

Store sparse matrices in special formats to avoid storing zeros

=> storage costs are reduced!

Avoid computing with zeros when working with sparse matrices.

=> MFlops needs are reduced!

Question: How do we represent sparse matrices and how do we compute with them?
Three Sparse Matrix Representations

**CRS**
- Indexed access to a row

**CCS**
- Indexed access to a column

**Co-ordinate Storage**
- Indexed access to neither rows nor columns
**MVM for CRS**

for $I = 1$ to $N$ do
  for $JJ = A$.rowptr($I$) to $A$.rowptr($I+1$) -1 do
    $Y(I) = Y(I) + A$.val($JJ$)*$X(A$.column($JJ$))
  od
od

**MVM for Co-ordinate storage**

for $P = 1$ to $NZ$ do
  $Y(A$.row($P$)) = $Y(A$.row($P$)) + $A$.val($P$)*$X(A$.column($P$))
od

Sparse matrix computations introduce subscripts with indirectness.
Computational Requirements with sparse matrices

- Assume $10^6$ mesh points (rows/columns of A).
- Assume roughly 100 non-zeros per row.
- Assume iterative solver needs 100 iterations to converge.
- Assume simulation runs for 1000 time steps.

One MVM requires roughly $10^8$ flops

$\Rightarrow$

Overall simulation requires $10^{13}$ flops and $10^8$ bytes of storage!

This is roughly 100 seconds on a 100 Gflop supercomputer.
Doable!
Flow-chart of Adaptive Finite-element Simulation of Fracture
Summary

- Computational science applications: solving pde’s or pushing particles
- PDE’s are solved using approximate techniques like fe method
- Time-consuming part: solving large linear algebraic systems
- Two approaches: iterative methods and direct (factorization) methods
- Key operations in iterative methods:
  Basic Linear Algebra Subroutines (BLAS)
  - Level-1 BLAS: inner-product of vectors, saxpy
  - Level-2 BLAS: matrix-vector product, triangular-solve
  - Level-3 BLAS: matrix-matrix product, triangular-solve with multiple right-hand-sides
- Important to exploit sparsity in matrix
- Exploiting sparsity complicates code.