Lecture 4:
Sources of Parallelism and Locality in Simulation

The HPC stack

- Applications
- Algorithms
- Software
- Hardware

Parallelism and Locality in Simulation

- Parallelism and data locality both critical to performance
  - Recall that moving data is the most expensive operation
- Real world problems have parallelism and locality:
  - Many objects operate independently of others.
  - Objects often depend much more on nearby than distant objects.
  - Dependence on distant objects can often be simplified.
    - Example of all three: particles moving under gravity
- Scientific models may introduce more parallelism:
  - When a continuous problem is discretized, time dependencies are generally limited to adjacent time steps.
    - Helps limit dependence to nearby objects (e.g., collisions)
  - Far-field effects may be ignored or approximated in many cases.
- Many problems exhibit parallelism at multiple levels

Basic Kinds of Simulation

- Discrete event systems:
    - "Game of Life," Manufacturing systems, Finance, Circuits, ...
- Particle systems:
    - Galaxies, Atoms, ...
- Lumped variables depending on continuous parameters
    - i.e., systems of Ordinary Differential Equations (ODEs),
    - Structural mechanics, Chemical kinetics, Circuits
- Continuous variables depending on continuous parameters
    - i.e., Partial Differential Equations (PDEs)
    - Heat, Elasticity, Electrostatics, Finance, Medical Image Analysis

- A given phenomenon can be modeled at multiple levels.
- Many simulations combine more than one of these techniques.

Discrete Event Systems

Discrete Event Systems

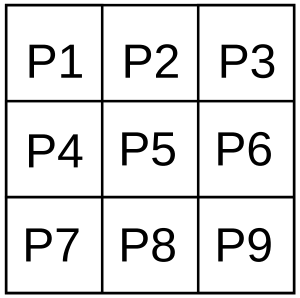
- Systems are represented as:
  - finite set of variables.
  - the set of all variable values at a given time is called the state.
  - each variable is updated by computing a transition function depending on some subset of the other variables.
- System may be:
  - synchronous: at each discrete timestep evaluate all transition functions; also called a state machine.
  - asynchronous: transition functions are evaluated only if the inputs change, based on an "event" from another part of the system; also called event driven simulation.
- Example: The "game of life:"
  - Space divided into cells, rules govern cell contents at each step

Conway's Game of Life

- The ***Game of Life***, also known simply as ***Life***, is a cellular automaton devised by the British mathematician John Horton Conway in 1970
- Zero-player game - evolution is determined by its initial state, no further input
- Universe is infinite 2D orthogonal grid of square cells.
- Each cell can be dead or alive
- Each cell interacts with its eight neighbors according to the rules:
    1. Any live cell with less than 2 live neighbors dies (underpopulation)
    2. Any live cell with 2 or 3 live neighbors lives
    3. Any live cell with more than 3 live neighbors dies (overpopulation)
    4. Any dead cell with 3 live neighbors becomes a live cell (reproduction)

Parallelism in Game of Life

- The simulation is synchronous
  - use two copies of the grid (old and new), "ping-pong" between them
  - the value of each new grid cell depends only on 9 cells (itself plus 8 neighbors) in old grid.
  - simulation proceeds in timesteps-- each cell is updated at every step.
- Easy to parallelize by dividing physical domain: *Domain Decomposition*

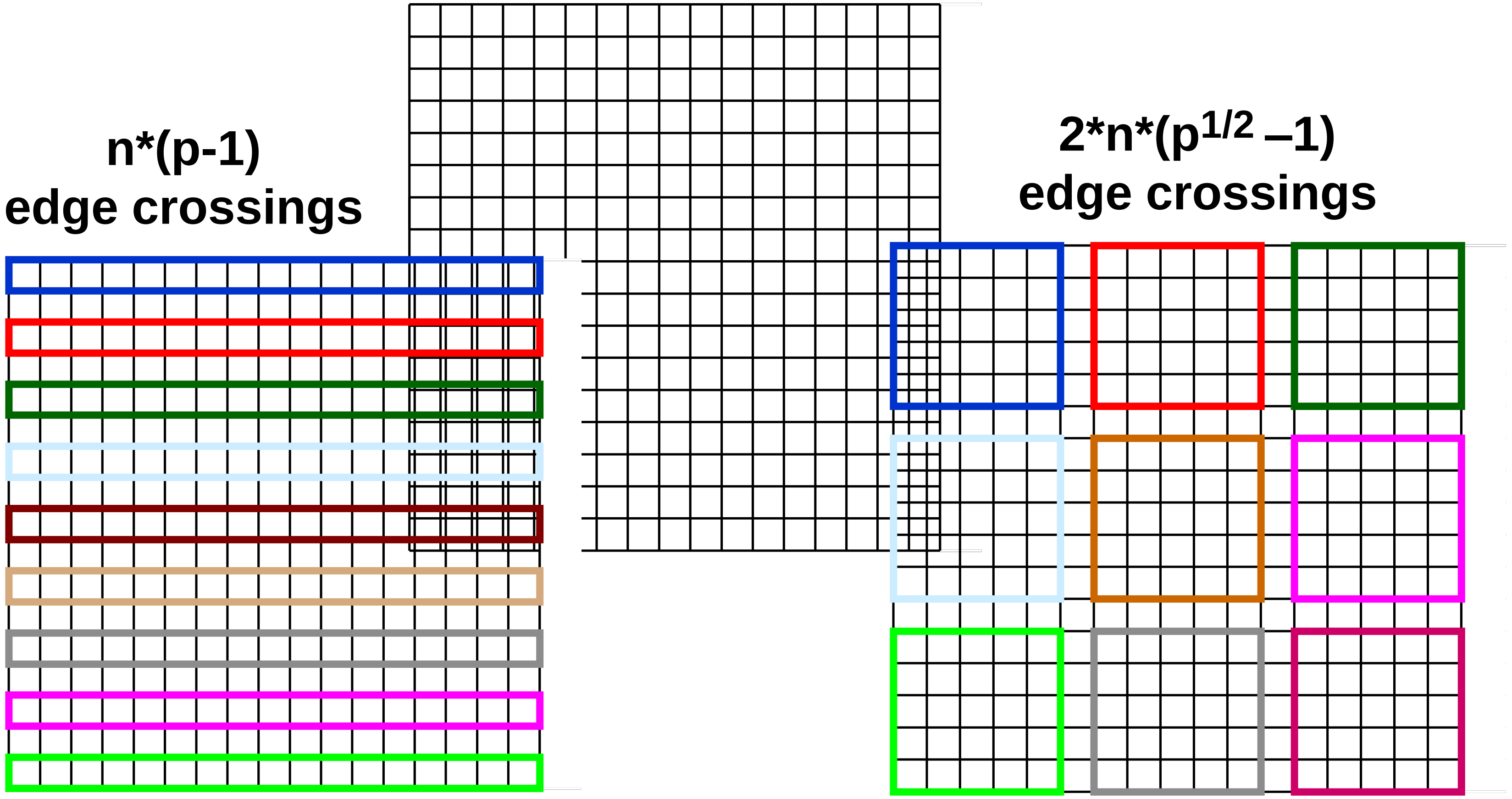


```
Repeat
    compute locally to update local system
    barrier()
    exchange state info with neighbors
    finish updates along border
until done simulating
```

- Locality is achieved by using large patches of the universe
  - Only boundary values from neighboring patches are needed.
- How to pick shapes of domains?

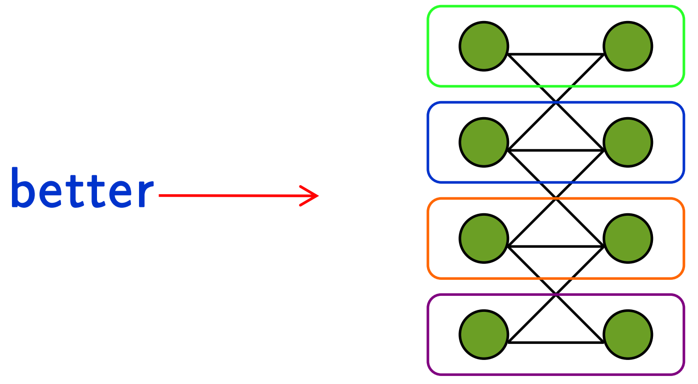
Regular Meshes (e.g. Game of Life)

- Suppose graph is nxn mesh with connection NSEW neighbors
- Which partition has less communication? (n=18, p=9)
- Minimizing communication on mesh ≡ minimizing "surface to volume ratio" of partition

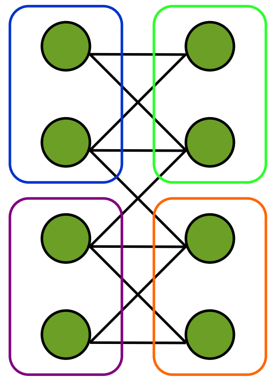


Graph Partitioning

- Graph partitioning assigns subgraphs to processors
  - Determines parallelism and locality.
  - Goal 1 is to evenly distribute subgraphs to nodes (load balance).
  - Goal 2 is to minimize edge crossings (minimize communication).
  - Easy for regular meshes, NP-hard in general, so we will approximate



edge crossings = 6



edge crossings = 10

Asynchronous Simulation

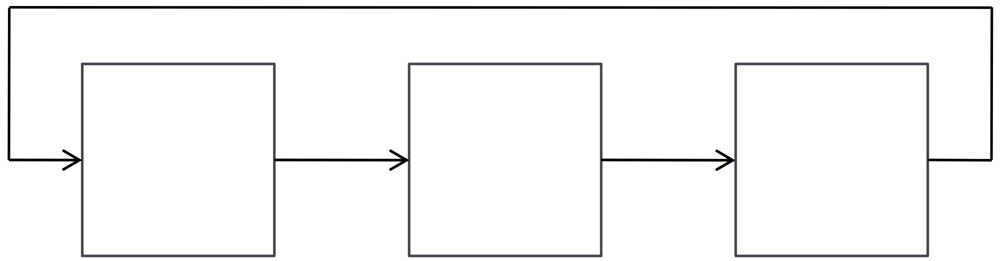
- Synchronous simulations may waste time:
  - Simulates even when the inputs do not change
- Asynchronous (event-driven) simulations update only when an event arrives from another component:
  - No global time steps, but individual events contain time stamp.
  - Example: Circuit simulation with delays (events are gates changing).
  - Example: Traffic simulation (events are cars changing lanes, etc.).
- Asynchronous is more efficient, but harder to parallelize
  - In MPI, events are naturally implemented as messages, but how do you know when to execute a "receive"?

Asynchronous Scheduling

- Conservative:
  - Only simulate up to (and including) the minimum time stamp of inputs.
  - Need deadlock detection if there are cycles in graph
    - Example on next slide
- Speculative (or Optimistic):
  - Assume no new inputs will arrive and keep simulating.
  - May need to backup if assumption wrong, using timestamps
- Optimizing load balance and locality is difficult:
  - Ex: circuit simulation
  - Locality means putting tightly coupled subcircuit on one processor.
  - Since "active" part of circuit likely to be in a tightly coupled subcircuit, this may be bad for load balance.

Deadlock in Conservative Asynchronous Circuit Simulation

- Example: processors simulating 3 ponds connected by streams along which fish can move



- Suppose all ponds simulated up to time t_0, but no fish move, so no messages sent from one proc to another
  - So no processor can simulate past time t_0
- Fix: After waiting for an incoming message for a while, send out an "Are you stuck too?" message
  - If you ever receive such a message, pass it on
  - If you receive such a message that you also sent, you have a deadlock cycle, so just take a step with latest input
- Can be a serial bottleneck

Summary of Discrete Event Simulations

- Model of the world is discrete
  - Both time and space

- Approaches
  - Decompose domain
    - graph partitioning problem
  - Run each component ahead using
    - Synchronous: communicate at end of each timestep
    - Asynchronous: communicate on-demand
      - Conservative scheduling – wait for inputs
        - need deadlock detection
      - Speculative scheduling – assume no inputs
        - roll back if necessary

Particle Systems

Particle Systems

- A particle system has
  - a finite number of particles
  - moving in space according to Newton's Laws (i.e., $F = ma$)
  - time is continuous
- Examples
  - stars in space with laws of gravity
  - electron beam in semiconductor manufacturing
  - atoms in a molecule with electrostatic forces
  - neutrons in a fission reactor
  - cars on a freeway with Newton's laws plus model of driver and engine
  - balls in a pinball game
- Note: many simulations combine techniques such as particle simulations with some discrete events

Forces in Particle Systems

- Force on each particle can be subdivided

$$\text{force} = \text{external\_force} + \text{nearby\_force} + \text{far\_field\_force}$$

- External force
  - ocean current in fish/pond simulation
  - externally imposed electric field in electron beam
- Nearby force
  - balls on a billiard table bounce off of each other
  - Van der Waals forces in fluid (1/r^6)
- Far-field force
  - gravity, electrostatics
  - forces governed by elliptic PDE

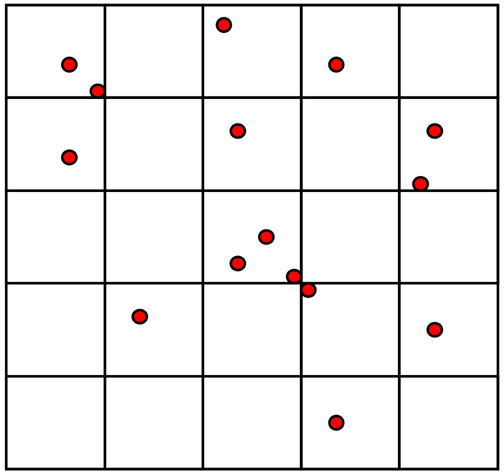
Parallelism in External Forces

- These are the simplest
- The force on each particle is independent
- Called "embarrassingly parallel"

- Evenly distribute particles on processors
  - Any distribution works
  - Locality is not an issue
- For each particle on processor, apply the external force

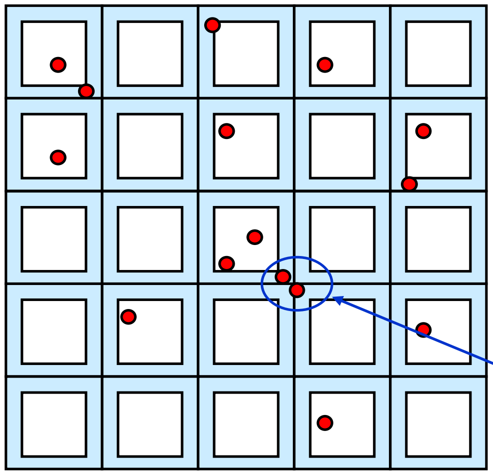
Parallelism in Nearby Forces

- Nearby forces require interaction and therefore communication.
- Force may depend on other nearby particles:
  - Example: collisions.
  - simplest algorithm is $O(n^2)$: look at all pairs to see if they collide.
- Usual parallel model is domain decomposition of physical region in which particles are located
  - $O(n/p)$ particles per processor if evenly distributed.



Parallelism in Nearby Forces

- Challenge 1: interactions of particles near processor boundary:
  - need to communicate particles near boundary to neighboring processors.
    - Region near boundary called "ghost zone"
  - Low surface to volume ratio means low communication.
    - Use squares, not slabs, to minimize ghost zone sizes

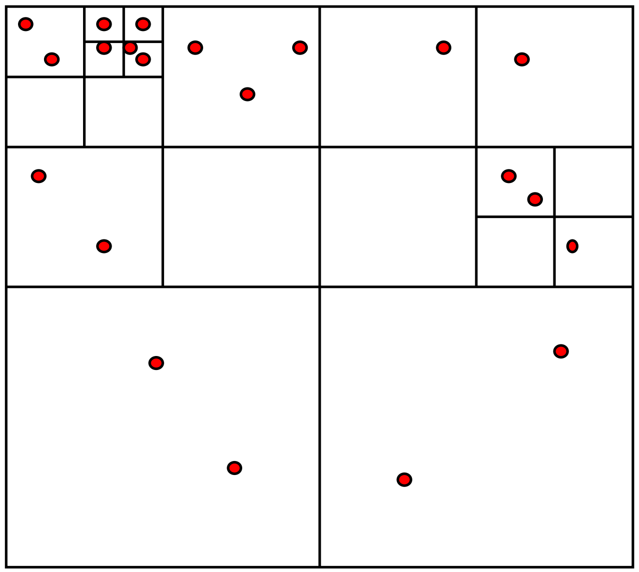


Communicate particles in boundary region to neighbors

Need to check for collisions between regions

Parallelism in Nearby Forces

- Challenge 2: load imbalance, if particles cluster:
  - galaxies, electrons hitting a device wall.
- To reduce load imbalance, divide space unevenly.
  - Each region contains roughly equal number of particles.
  - Quad-tree in 2D, oct-tree in 3D.

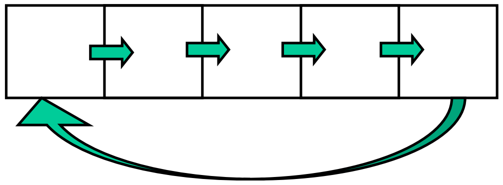


Example: each square contains at most 3 particles

- May need to rebalance as particles move, hopefully seldom

Parallelism in Far-Field Forces

- Far-field forces involve all-to-all interaction and therefore communication.
- Force depends on all other particles:
  - Examples: gravity, protein folding
  - Simplest algorithm is O(n^2)
  - Just decomposing space does not help since every particle needs to "visit" every other particle.



Implement by rotating particle sets.

- Keeps processors busy
- All processors eventually see all particles

- Use more clever algorithms to reduce communication
- Use more clever algorithms to beat O(n^2).

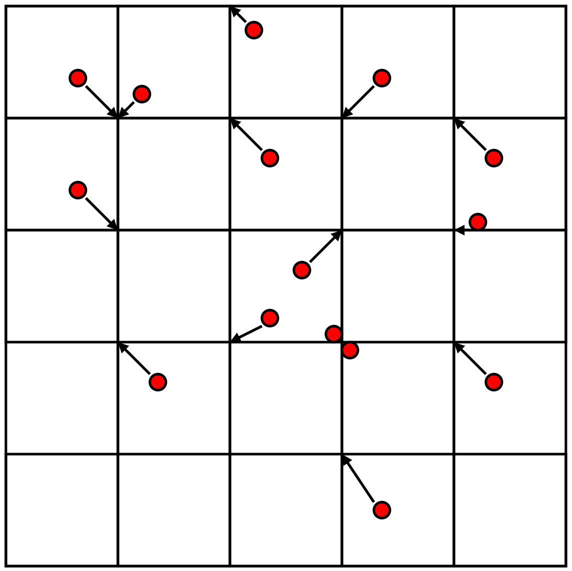
Far-field Forces: Particle-Mesh Methods

- Based on approximation:
  - Superimpose a regular mesh.
  - "Move" particles to nearest grid point.
- Exploit fact that the far-field force satisfies a PDE that is easy to solve on a regular mesh:
  - FFT, multigrid (described in future lectures)
  - Cost drops to O(n log n) or O(n) instead of O(n^2)
- Accuracy depends on the fineness of the grid is and the uniformity of the particle distribution.

1) Particles are moved to nearby mesh points

2) Solve mesh problem

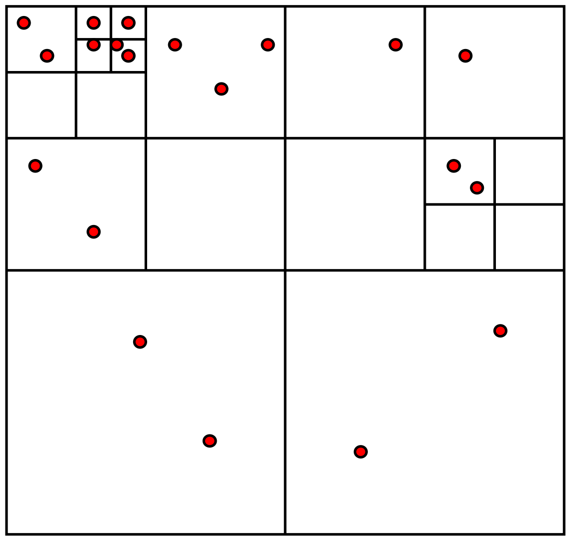
3) Forces are interpolated at particles from mesh points



Far-field forces: Tree Decomposition

- Based on approximation.
  - Forces from group of far-away particles "simplified" -- resembles a single large particle.
  - Use tree; each node contains an approximation of descendants.
- Also O(n log n) or O(n) instead of O(n^2).

- Several Algorithms
  - Barnes-Hut
  - Fast multipole method (FMM) of Greengard/Rohklin
  - Anderson's method



Summary of Particle Methods

- Model contains discrete entities, namely, particles
- Time is continuous – must be discretized to solve

- Simulation follows particles through timesteps
  - Force = external _force + nearby_force + far_field_force
  - All-pairs algorithm is simple, but inefficient, O(n^2)
  - Particle-mesh methods approximates by moving particles to a regular mesh, where it is easier to compute forces
  - Tree-based algorithms approximate by treating set of particles as a group, when far away

- May think of this as a special case of a "lumped" system

Lumped Systems: ODEs

System of Lumped Variables

- Many systems are approximated by
  - System of "lumped" variables.
  - Each depends on continuous parameter (usually time).
- Example -- circuit:
  - approximate as graph.
    - wires are edges.
    - nodes are connections between 2 or more wires.
    - each edge has resistor, capacitor, inductor or voltage source.
  - system is "lumped" because we are not computing the voltage/current at every point in space along a wire, just endpoints.
  - Variables related by Ohm's Law, Kirchoff's Laws, etc.
- Forms a system of ordinary differential equations (ODEs).
  - Differentiated with respect to time
  - Variant: ODEs with some constraints
    - Also called DAEs, Differential Algebraic Equations

Circuit Example

- State of the system is represented by
  - $v_n(t)$ node voltages
  - $i_b(t)$ branch currents
  - $v_b(t)$ branch voltages

$$\left.\begin{array}{l} v_n(t) \\ i_b(t) \\ v_b(t) \end{array}\right\} \text{all at times } t$$

- Equations include
  - Kirchoff's current
  - Kirchoff's voltage
  - Ohm's law
  - Capacitance
  - Inductance

$$\begin{pmatrix} 0 & A & 0 \\ A' & 0 & -I \\ 0 & R & -I \\ 0 & -I & C*d/dt \\ 0 & L*d/dt & I \end{pmatrix} * \begin{pmatrix} v_n \\ i_b \\ v_b \end{pmatrix} = \begin{pmatrix} 0 \\ S \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

- A is sparse matrix, representing connections in circuit
  - One column per branch (edge), one row per node (vertex) with +1 and -1 in each column at rows indicating end points
- Write as single large system of ODEs or DAEs

Structural Analysis Example

- Another example is structural analysis in civil engineering:
  - Variables are displacement of points in a building.
  - Newton's and Hook's (spring) laws apply.
  - Static modeling: exert force and determine displacement.
  - Dynamic modeling: apply continuous force (earthquake).
  - Eigenvalue problem: do the resonant modes of the building match an earthquake?

Solving ODEs

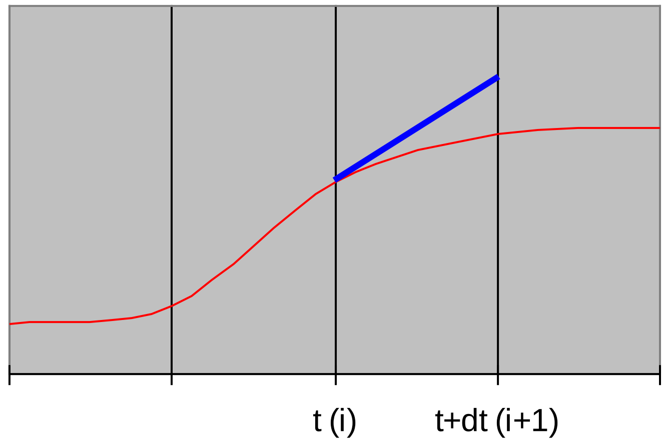
- In these examples, and most others, the matrices are sparse:
  - i.e., most array elements are 0.
  - neither store nor compute on these 0's.
  - Sparse because each component only depends on a few others

- Given a set of ODEs, two kinds of questions are:
  - Compute the values of the variables at some time t
    - Explicit methods
    - Implicit methods
  - Compute modes of vibration
    - Eigenvalue problems

Solving ODEs: Explicit Methods

- Assume ODE is x'(t) = f(x) = A x(t), where A is a sparse matrix
  - Compute x(i*dt) = x[i]
    at i=0,1,2,…
  - ODE gives x'(i*dt) = slope
    x[i+1] = x[i] + dt*slope

Use slope at x[i]

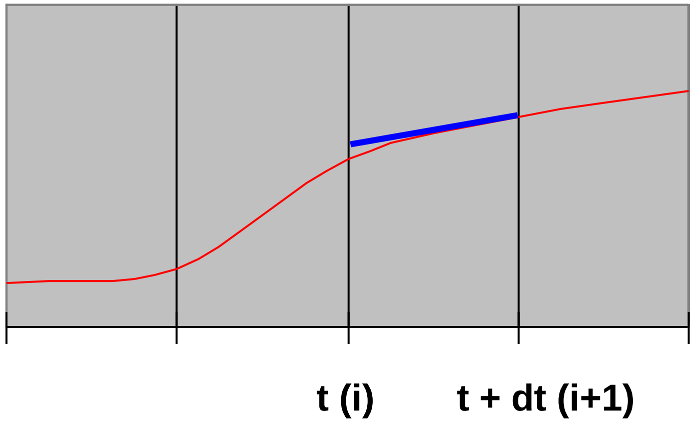


- Explicit methods, e.g., (Forward) Euler's method.
  - Approximate x'(t)=A x(t) by (x[i+1] - x[i])/dt = A x[i].
  - x[i+1] = x[i]+dt *A x[i], i.e., sparse matrix-vector multiplication.
- Tradeoffs:
  - Simple algorithm: sparse matrix vector multiply.
  - Stability problems: May need to take very small time steps, especially if system is stiff (i.e. A has some large entries, so x can change rapidly).

Solving ODEs: Implicit Methods

- Assume ODE is x'(t) = f(x) = A x(t) , where A is a sparse matrix
  - Compute x(i*dt) = x[i]
    at i=0,1,2,…
  - ODE gives x'((i+1)*dt) = slope
    x[i+1] = x[i] + dt*slope

  Use slope at x[i+1]



- Implicit method, e.g., Backward Euler solve:
  - Approximate x'(t)=A x(t) by (x[i+1] - x[i])/dt = A x[i+1].
  - (I - dt*A) x[i+1] = x[i], i.e. we need to solve a sparse linear system of equations.
- Trade-offs:
  - Larger timestep possible: especially for stiff problems
  - More difficult algorithm: need to solve a sparse linear system of equations at each step

Solving ODEs: Eigensolvers

- Computing modes of vibration: finding eigenvalues and eigenvectors.
  - Seek solution of $d^2x(t)/dt^2 = A\,x(t)$ of form

$$x(t) = \sin(\omega t)\, x_0,$$

where x_0 is a constant vector

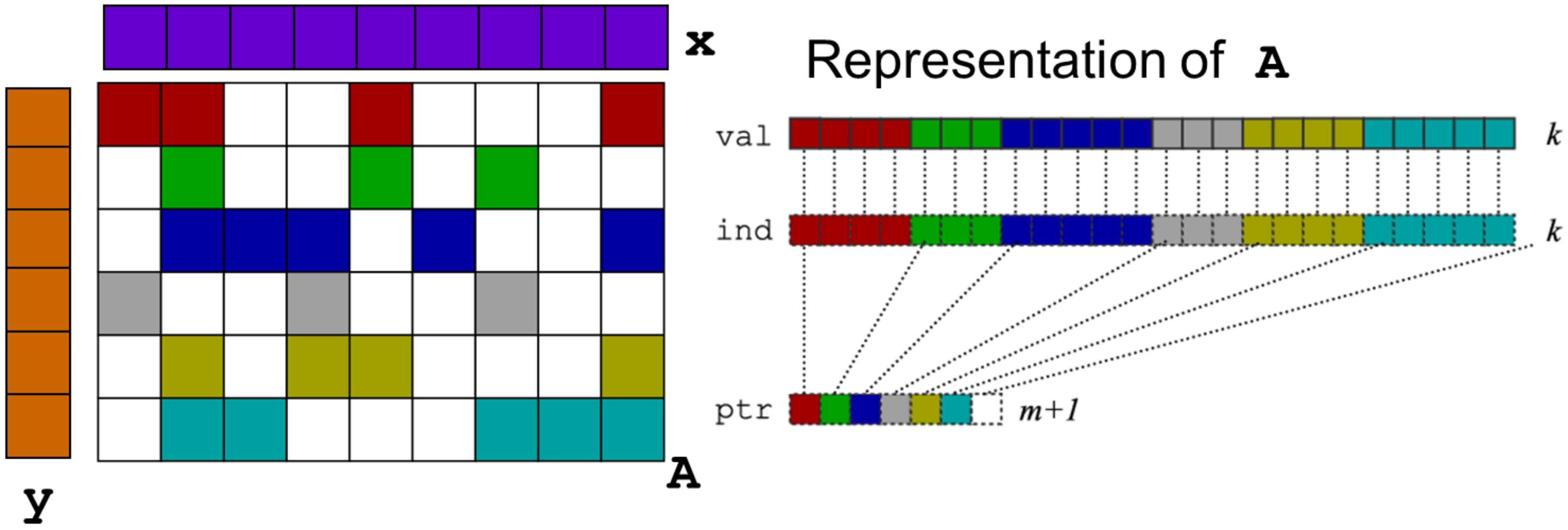
    - ω called the frequency of vibration
    - x_0 sometimes called a "mode shape"
  - Plug in to get $-\omega^2 x_0 = Ax_0$, so that $-\omega^2$ is an eigenvalue and x_0 is an eigenvector of A.
  - Solution schemes reduce either to sparse-matrix multiplications, or solving sparse linear systems.

Summary of ODE Methods

- Explicit methods for ODEs need sparse-matrix-vector mult.
- Implicit methods for ODEs need to solve linear systems
- Direct methods (Gaussian elimination)
  - Called LU Decomposition, because we factor $A = LU$.
  - Future lectures will consider both dense and sparse cases.
  - More complicated than sparse-matrix vector multiplication.
- Iterative solvers
  - Will discuss several of these in future.
    - Jacobi, Successive over-relaxation (SOR) , Conjugate Gradient (CG), Multigrid,...
  - Most have sparse-matrix-vector multiplication in kernel.
- Eigenproblems
  - Future lectures will discuss dense and sparse cases.
  - Also depend on sparse-matrix-vector multiplication, direct methods.

SpMV in Compressed Sparse Row (CSR) Format

SpMV: y = y + A*x, only store, do arithmetic, on nonzero entries
CSR format is simplest one of many possible data structures for A

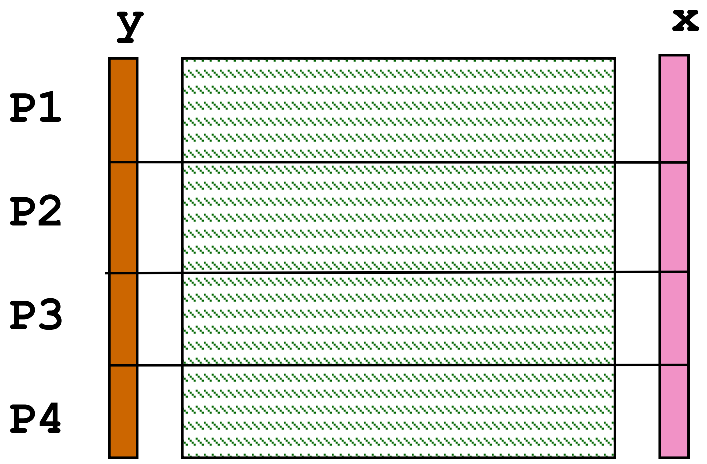


Matrix-vector multiply kernel: y(i) ← y(i) + A(i,j) x(j)

```
for each row i
  for k=ptr[i] to ptr[i+1]-1 do
      y[i] = y[i] + val[k]*x[ind[k]]
```

Parallel Sparse Matrix-vector multiplication

- y = Ax, where A is a sparse n x n matrix



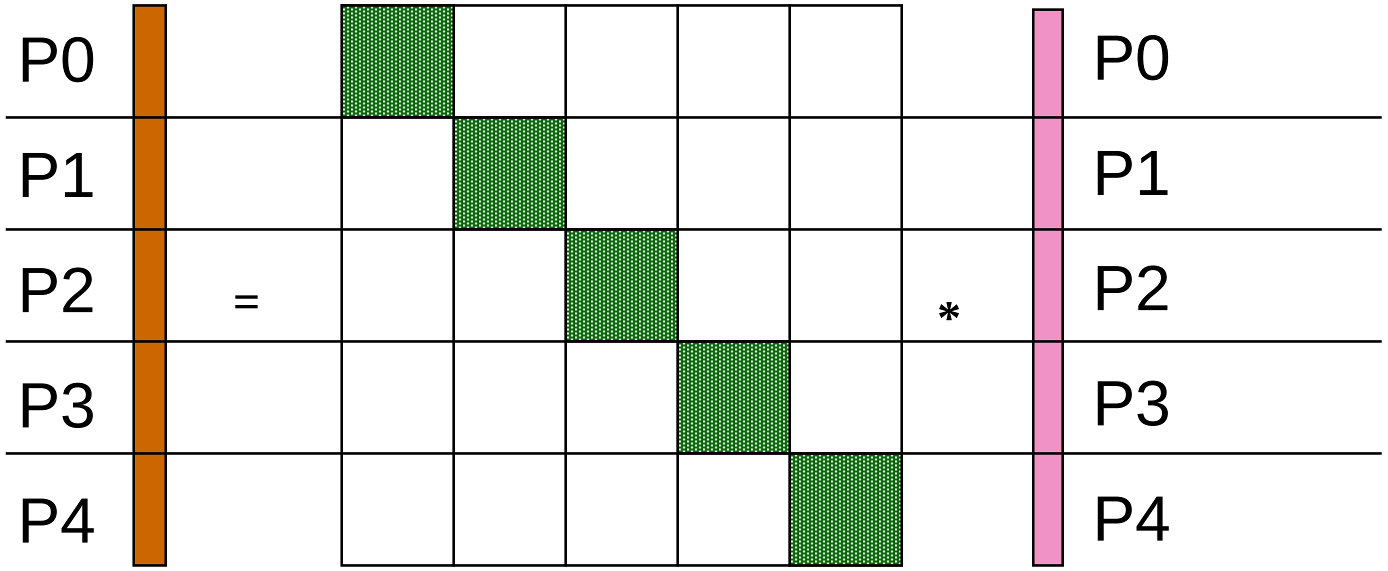
- Questions
  - which processors store
    - y[i], x[i], and A[i,j]
  - which processors compute
    - y[i] = sum (from 1 to n) A[i,j] * x[j]

      = (row i of A) * x ... a sparse dot product
- Partitioning
  - Partition index set {1,…,n} = N1 ∪ N2 ∪ … ∪ Np.
  - For all i in Nk, Processor k stores y[i], x[i], and row i of A
  - For all i in Nk, Processor k computes y[i] = (row i of A) * x
    - "owner computes" rule: Processor k compute the y[i]'s it owns.

May require communication

Matrix Reordering via Graph Partitioning

- "Ideal" matrix structure for parallelism: block diagonal
  - p (number of processors) blocks, can all be computed locally.
  - If no non-zeros outside these blocks, no communication needed
- Can we reorder the rows/columns to get close to this?
  - Most nonzeros in diagonal blocks, few outside



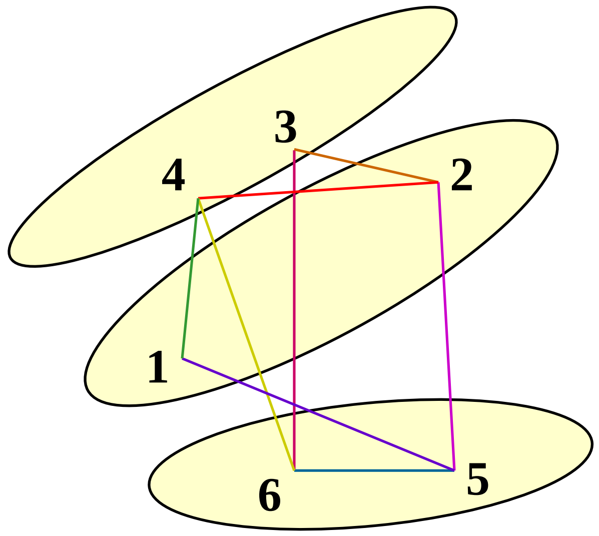
Goals of Reordering

- Performance goals
  - balance load (how is load measured?)
    - Approx equal number of nonzeros (not necessarily rows)
  - balance storage (how much does each processor store?)
    - Approx equal number of nonzeros
  - minimize communication (how much is communicated?)
    - Minimize nonzeros outside diagonal blocks
    - Related optimization criterion is to move nonzeros near diagonal
  - improve register and cache re-use
    - Group nonzeros in small vertical blocks so source (x) elements loaded into cache or registers may be reused (temporal locality)
    - Group nonzeros in small horizontal blocks so nearby source (x) elements in the cache may be used (spatial locality)
- Other algorithms reorder for other reasons
  - Reduce # nonzeros in matrix after Gaussian elimination
  - Improve numerical stability

Graph Partitioning and Sparse Matrices

- Relationship between matrix and graph

$$\begin{array}{c|cc|cc|cc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 1 & & & 1 & 1 & \\
2 & & 1 & 1 & 1 & 1 & \\
\hline
3 & & 1 & 1 & & & 1 \\
4 & 1 & 1 & & 1 & & 1 \\
\hline
5 & 1 & 1 & & & 1 & 1 \\
6 & & & 1 & 1 & 1 & 1 \\
\hline
\end{array}$$



- Edges in the graph are nonzero in the matrix: here the matrix is symmetric (edges are unordered) and weights are equal (1)
- If divided over 3 procs, there are 14 nonzeros outside the diagonal blocks, which represent the 7 (bidirectional) edges

Summary: Common Problems

- Load balancing
  - May be due to lack of parallelism or poor work distribution
  - Statically, divide grid (or graph) into blocks
  - Dynamically, if load changes significantly during run
- Locality
  - Partition into large chunks with low surface-to-volume ratio
    - To minimize communication
  - Distributed particles according to location, but use irregular spatial decomposition (e.g., quad tree) for load balance
- Constant tension between these two
  - Particle-Mesh method: can't balance particles (moving), balance mesh (fixed) and keep particles near mesh points without communication
- Linear algebra
  - Solving linear systems (sparse and dense)
  - Eigenvalue problems will use similar techniques
- Fast Particle Methods
  - O(n log n) instead of O(n^2)

Partial Differential Equations PDEs

Continuous Variables, Continuous Parameters

Examples of such systems include:

- Elliptic problems (steady state, global space dependence)
    - Electrostatic or Gravitational Potential: Potential(position)
- Hyperbolic problems (time dependent, local space dependence):
    - Sound waves: Pressure(position,time)
- Parabolic problems (time dependent, global space dependence)
    - Heat flow: Temperature(position, time)
    - Diffusion: Concentration(position, time)

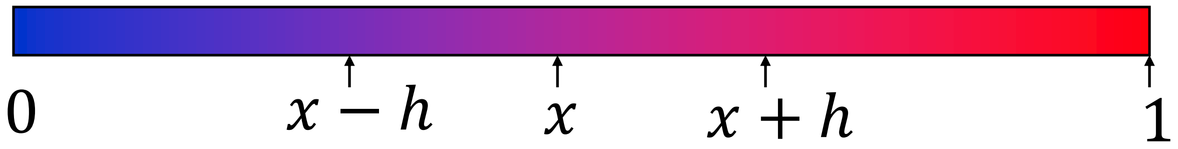
Global vs Local Dependence

- Global means either a lot of communication, or tiny time steps
- Local arises from finite wave speeds: limits communication

Many problems combine features of above

- Fluid flow: Velocity,Pressure,Density(position,time)
- Elasticity: Stress,Strain(position,time)

Example: Deriving the Heat Equation



Consider a simple problem

- A bar of uniform material, insulated except at ends
- Let $u(x,t)$ be the temperature at position x at time t
- Heat travels from $x-h$ to $x+h$ at rate proportional to:

$$\frac{d\ u(x,t)}{dt} = C \quad \frac{(u(x\text{-}h,t)\text{-}u(x,t))/h\ \text{-}\ (u(x,t)\text{-}\ u(x+h,t))/h}{h}$$

As h → 0, we get the heat equation:

$$\frac{d\ u(x,t)}{dt} = C \quad \frac{d^2\ u(x,t)}{dx^2}$$

Details of the Explicit Method for Heat

$$\frac{d\ u(x,t)}{dt} = C\ \frac{d^2\ u(x,t)}{dx^2}$$

- Discretize time and space using explicit approach (forward Euler) to approximate time derivative:

$$(u(x,t+\delta) - u(x,t))/\delta \ = \ C\ [\ (u(x\text{-}h,t)\text{-}u(x,t))/h - (u(x,t)\text{-}\ u(x+h,t))/h\ ]\ /\ h$$
$$= \ C\ [u(x\text{-}h,t) - 2*u(x,t) + u(x+h,t)]/h^2$$

Solve for $u(x,t+\delta)$:

$$u(x,t+\delta) = \ u(x,t)+ C*\delta\ /h^2\ *(u(x\text{-}h,t) - 2*u(x,t) + u(x+h,t))$$

- Let $z = C*\delta\ /h^2$, simplify:

$$u(x,t+\delta) = \ z*\ u(x\text{-}h,t) + (1\text{-}2z)*u(x,t) + z*u(x+h,t)$$

- Change variable x to $j*h$, t to $i*\delta$, and $u(x,t)$ to $u[j,i]$

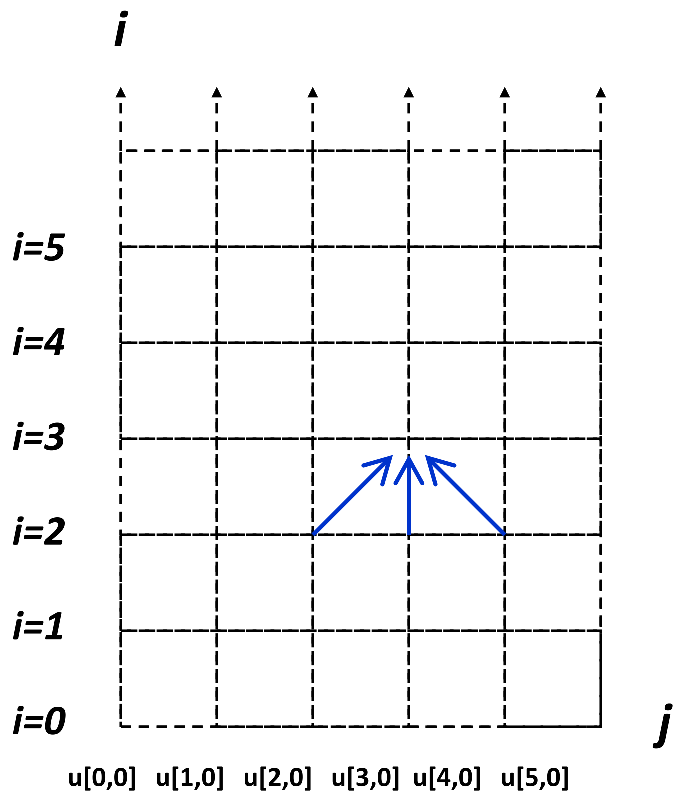
$$u[j,i+1]= z*u[j\text{-}1,i]+ (1\text{-}2*z)*u[j,i]+ z*u[j+1,i]$$

Explicit Solution of the Heat Equation

- Use "finite differences" with u[j,i] as the temperature at
  - time $t = i*\delta$ (i = 0,1,2,...) and position x = j*h (j=0,1,…,N=1/h)
  - initial conditions on u[j,0]
  - boundary conditions on u[0,i] and u[N,i]
- At each timestep i = 0,1,2,...

```
For j=1 to N-1
    u[j,i+1]= z*u[j-1,i]+ (1-2*z)*u[j,i] + z*u[j+1,i]
where z =C*δ/h²
```

- This corresponds to
  - Matrix-vector-multiply by T (next slide)
  - Combine nearest neighbors on grid



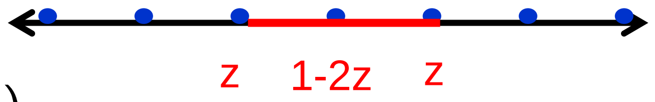
Matrix View of Explicit Method for Heat

u[j,i+1]= z*u[j-1,i]+ (1-2*z)*u[j,i] + z*u[j+1,i],   same as:

u[:,i+1] = T * u[:,i] where T is tridiagonal:

$$T = \begin{pmatrix} 1-2z & z & & & \\ z & 1-2z & z & & \\ & z & 1-2z & z & \\ & & z & 1-2z & z \\ & & & z & 1-2z \end{pmatrix} = I - zL, \quad L = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

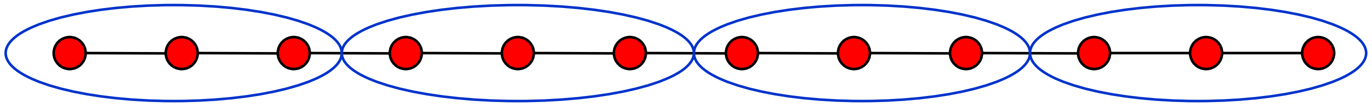
Graph and "3 point stencil"



- L called Laplacian (in 1D)
- For a 2D mesh (5 point stencil) the Laplacian is pentadiagonal
  - More on the matrix/grid views later

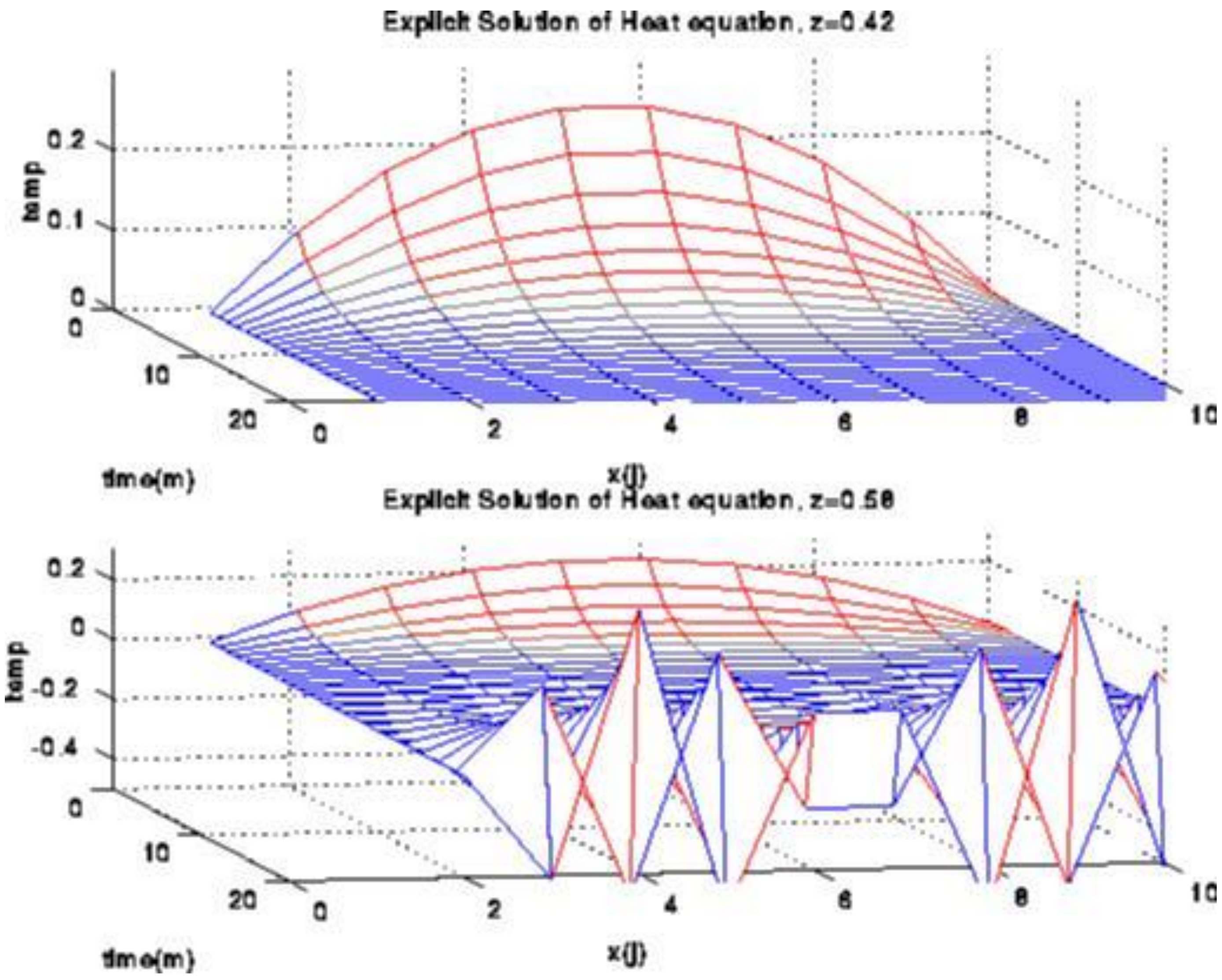
Parallelism in Explicit Method for PDEs

- Sparse matrix vector multiply, via Graph Partitioning
- Partitioning the space (x) into p chunks
  - good load balance (assuming large number of points relative to p)
  - minimize communication (least dependence on data outside chunk)



- Generalizes to
  - multiple dimensions.
  - arbitrary graphs (= arbitrary sparse matrices).
- Explicit approach often used for hyperbolic equations
  - Finite wave speed, so only depend on nearest chunks
- Problem with explicit approach for heat (parabolic):
  - numerical instability.
  - solution blows up eventually if $z = C\delta/h^2 > .5$
  - need to make the time step δ very small when h is small: $\delta < .5*h^2 /C$

Instability in Solving the Heat Equation Explicitly



Implicit Solution of the Heat Equation

$$\frac{d\ u(x,t)}{dt} = C\ \frac{d^2\ u(x,t)}{dx^2}$$

- Discretize time and space using implicit approach (Backward Euler) to approximate time derivative:

$$(u(x,t+\delta) - u(x,t))/dt = C*(u(x-h,t+\delta) - 2*u(x,t+\delta) + u(x+h,\ t+\delta))/h^2$$

$$u(x,t) = u(x,t+\delta) - C*\delta/h^2 *(u(x-h,t+\delta) - 2*u(x,t+\delta) + u(x+h,t+\delta))$$

- Let $z = C*\delta/h^2$ and change variable t to $i*\delta$, x to $j*h$ and $u(x,t)$ to $u[j,i]$

$$(I + zL)*\ u[:,\ i+1] = u[:,i]$$

- Where I is identity and

  L is Laplacian as before

$$\mathbf{L} = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

Implicit Solution of the Heat Equation

- The previous slide derived Backward Euler

$$(I + zL) * u[:, i+1] = u[:,i]$$

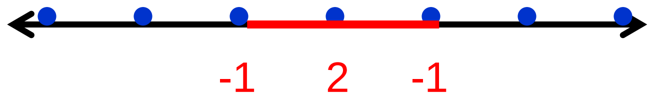
- But the Trapezoidal Rule has better numerical properties:

$$(I + (z/2)L) * u[:,i+1] = (I - (z/2)L) * u[:,i]$$

- Again **I** is the identity matrix and **L** is:

$$\mathbf{L} = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

Graph and "stencil"



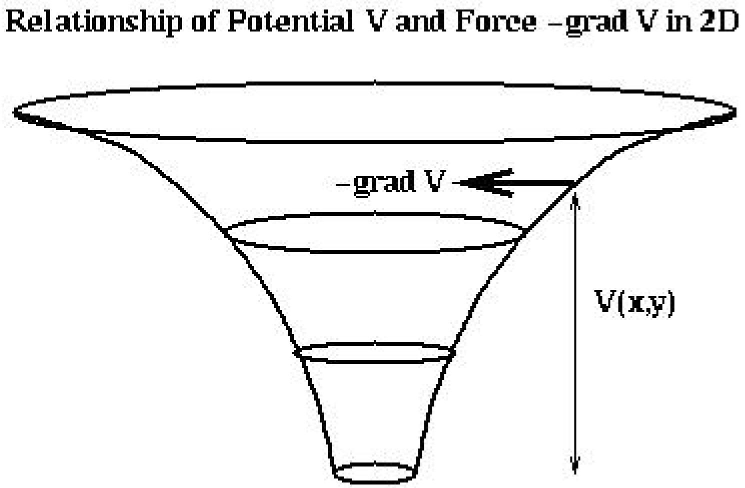
- Other problems (elliptic instead of parabolic) yield Poisson's equation (Lx = b in 1D)

Relation of Poisson to Gravity, Electrostatics

- Poisson equation arises in many problems
- E.g., force on particle at (x,y,z) due to particle at 0 is

  $-(x,y,z)/r^3$, where $r = \mathrm{sqrt}(x^2 + y^2 + z^2)$
- Force is also gradient of potential V = -1/r

  = -(d/dx V, d/dy V, d/dz V) = -grad V
- V satisfies Poisson's equation



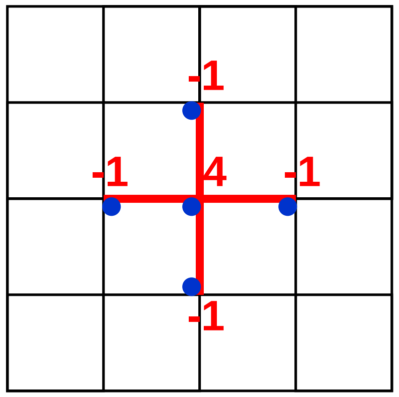
$$\frac{d^2V}{dx^2} + \frac{d^2V}{dy^2} + \frac{d^2V}{dz^2} = 0$$

2D Implicit Method

- Similar to the 1D case, but the matrix *L* is now

$$
L = \left(\begin{array}{ccc|ccc|ccc}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
& -1 & 4 & & & -1 & & & \\
\hline
-1 & & & 4 & -1 & & -1 & & \\
& -1 & & -1 & 4 & -1 & & -1 & \\
& & -1 & & -1 & 4 & & & -1 \\
\hline
& & & -1 & & & 4 & -1 & \\
& & & & -1 & & -1 & 4 & -1 \\
& & & & & -1 & & -1 & 4
\end{array}\right)
$$

Graph and "5 point stencil"

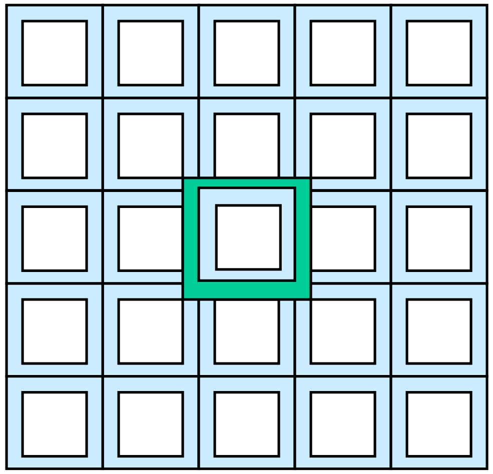


3D case is analogous (7 point stencil)

- Multiplying by this matrix (as in the explicit case) is simply nearest neighbor computation on 2D grid.
- To solve this system, there are several techniques.

Parallelism in Regular meshes

- Computing a Stencil on a regular mesh
  - need to communicate mesh points near boundary to neighboring processors.
    - Often done with ghost regions
  - Surface-to-volume ratio keeps communication down, but
    - Still may be problematic in practice



Implemented using "ghost" regions.

Adds memory overhead

Overview of Algorithms

- Sorted in two orders (roughly):
  - from slowest to fastest on sequential machines.
  - from most general (works on any matrix) to most specialized (works on matrices "like" T).
- Dense LU: Gaussian elimination; works on any N-by-N matrix.
- Band LU: Exploits the fact that T is nonzero only on sqrt(N) diagonals nearest main diagonal.
- Jacobi: Essentially does matrix-vector multiply by T in inner loop of iterative algorithm.
- Explicit Inverse: Assume we want to solve many systems with T, so we can precompute and store inv(T) "for free", and just multiply by it (but still expensive).
- Conjugate Gradient: Uses matrix-vector multiplication, like Jacobi, but exploits mathematical properties of T that Jacobi does not.
- Red-Black SOR (successive over-relaxation): Variation of Jacobi that exploits yet different mathematical properties of T. Used in multigrid schemes.
- Sparse LU: Gaussian elimination exploiting particular zero structure of T.
- FFT (Fast Fourier Transform): Works only on matrices *very* like T.
- Multigrid: Also works on matrices like T, that come from elliptic PDEs.
- Lower Bound: Serial (time to print answer); parallel (time to combine N inputs).

Algorithms for 2D (3D) Poisson Equation (N vars)

Algorithm	Serial	PRAM	Memory	#Procs
• Dense LU	N^3	N	N^2	N^2
• Band LU	N^2 $(N^{7/3})$	N	$N^{3/2}$ $(N^{5/3})$	$N(N^{4/3})$
• Jacobi	N^2 $(N^{5/3})$	N $(N^{2/3})$	N	N
• Explicit Inv.	N^2	$\log N$	N^2	N^2
• Red/Black SOR	$N^{3/2}$ $(N^{4/3})$	$N^{1/2}$ $(N^{4/3})$	N	N
• Sparse LU	$N^{3/2}$ (N^2)	$N^{1/2}$ $(N^{2/3})$	$N*\log N$ $(N^{4/3})$	$N(N^{4/3})$
• FFT	$N*\log N$	$\log N$	N	N
• Multigrid	N	$\log^2 N$	N	N
• Lower bound	N	$\log N$	N	

All entries in "Big-Oh" sense (constants omitted)
PRAM is an idealized parallel model with zero cost communication

Summary of Approaches to Solving PDEs

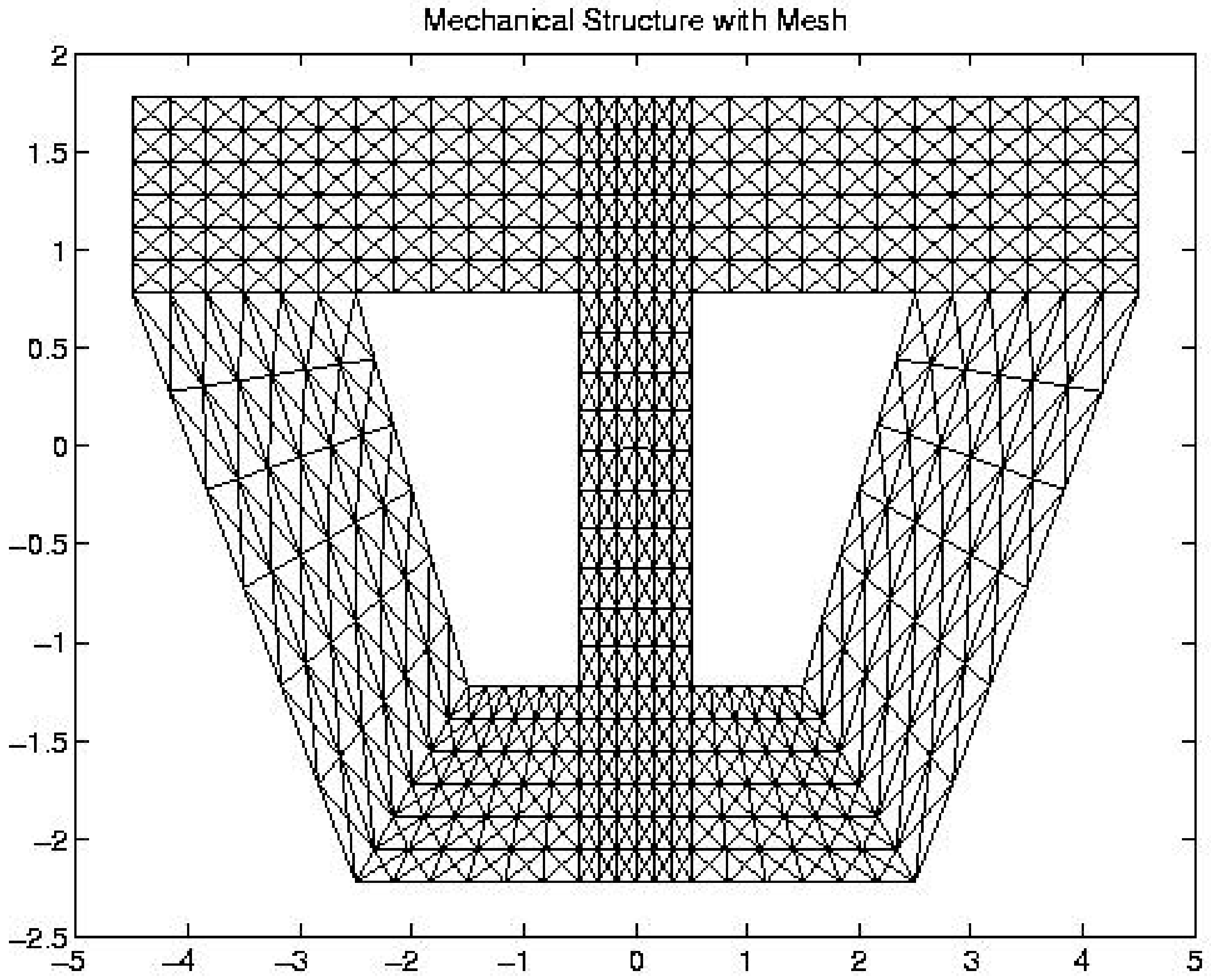
- As with ODEs, either explicit or implicit approaches are possible
  - Explicit, sparse matrix-vector multiplication
  - Implicit, sparse matrix solve at each step
    - Direct solvers are hard (more on this later)
    - Iterative solves turn into sparse matrix-vector multiplication
      - Graph partitioning

- Graph and sparse matrix correspondence:
  - Sparse matrix-vector multiplication is nearest neighbor "averaging" on the underlying mesh
- Not all nearest neighbor computations have the same efficiency
  - Depends on the mesh structure (nonzero structure) and the number of flops per point.

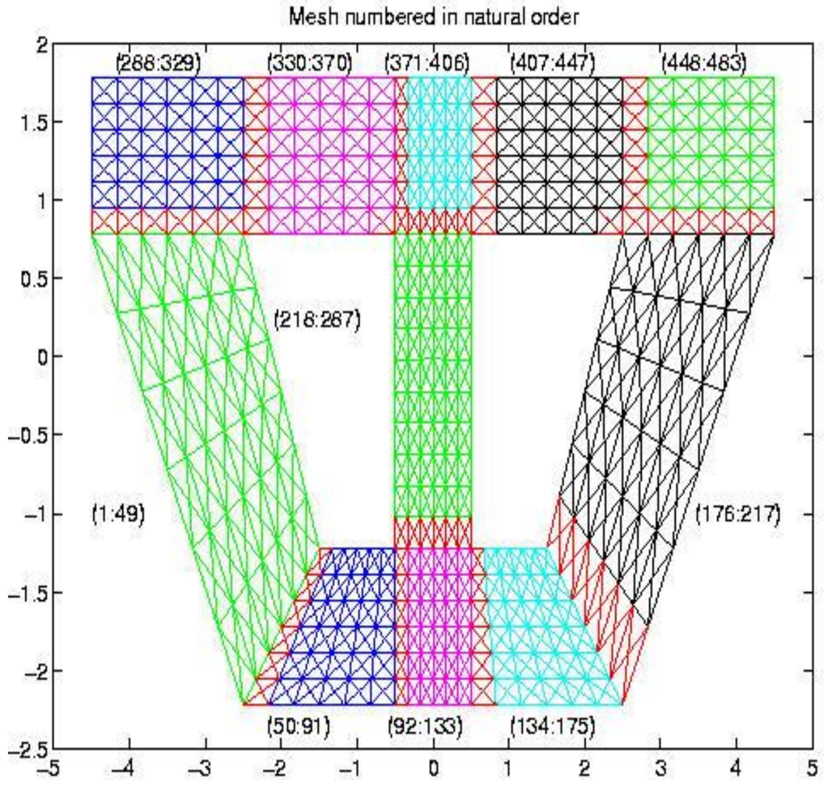
Comments on practical meshes

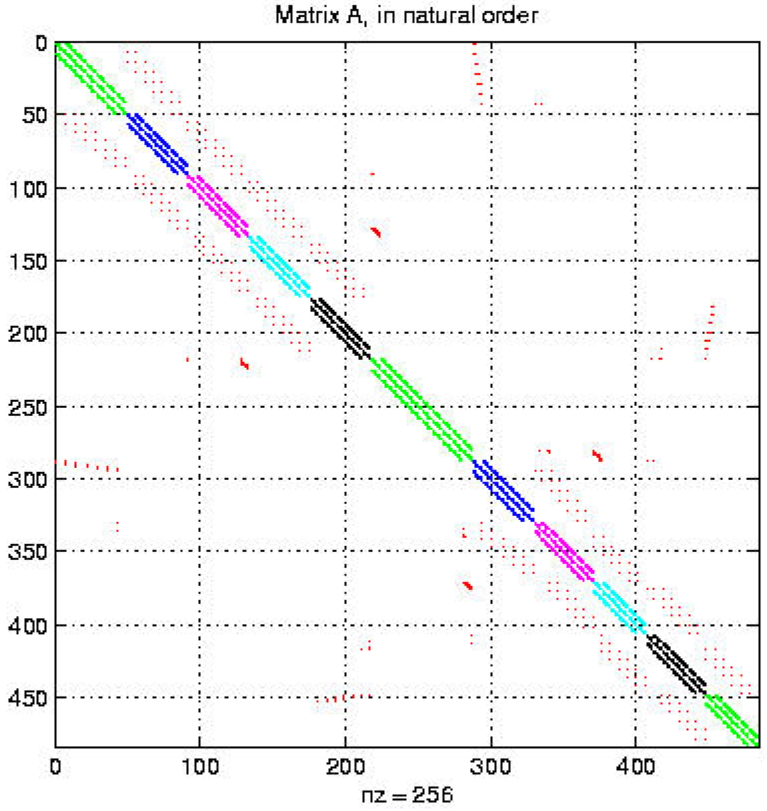
- Regular 1D, 2D, 3D meshes
  - Important as building blocks for more complicated meshes
- Practical meshes are often irregular
  - Composite meshes, consisting of multiple "bent" regular meshes joined at edges
  - Unstructured meshes, with arbitrary mesh points and connectivities
  - Adaptive meshes, which change resolution during solution process to put computational effort where needed

Composite mesh from a mechanical structure



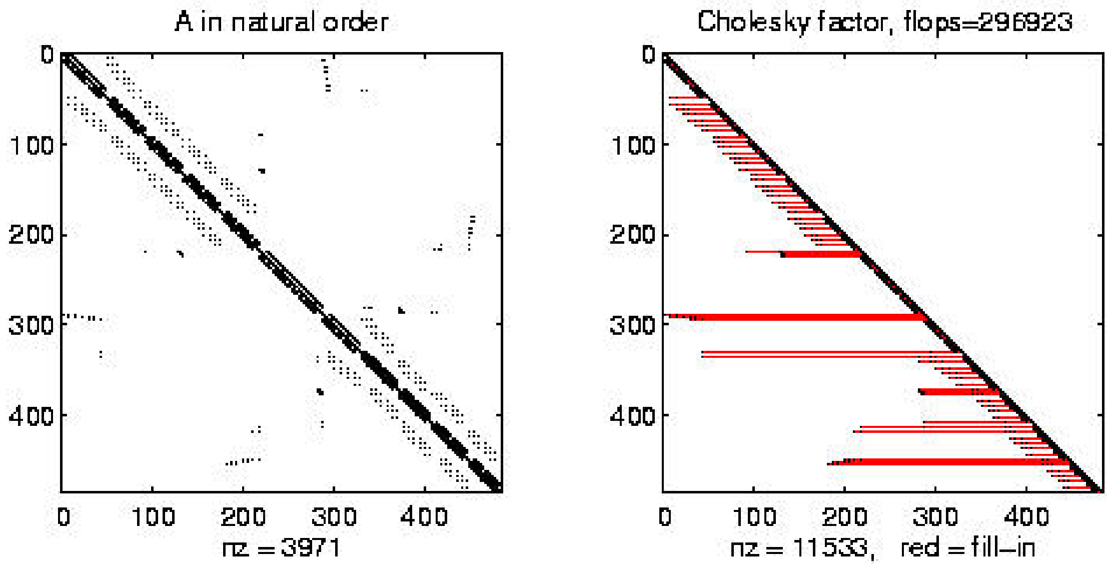
Converting the mesh to a matrix



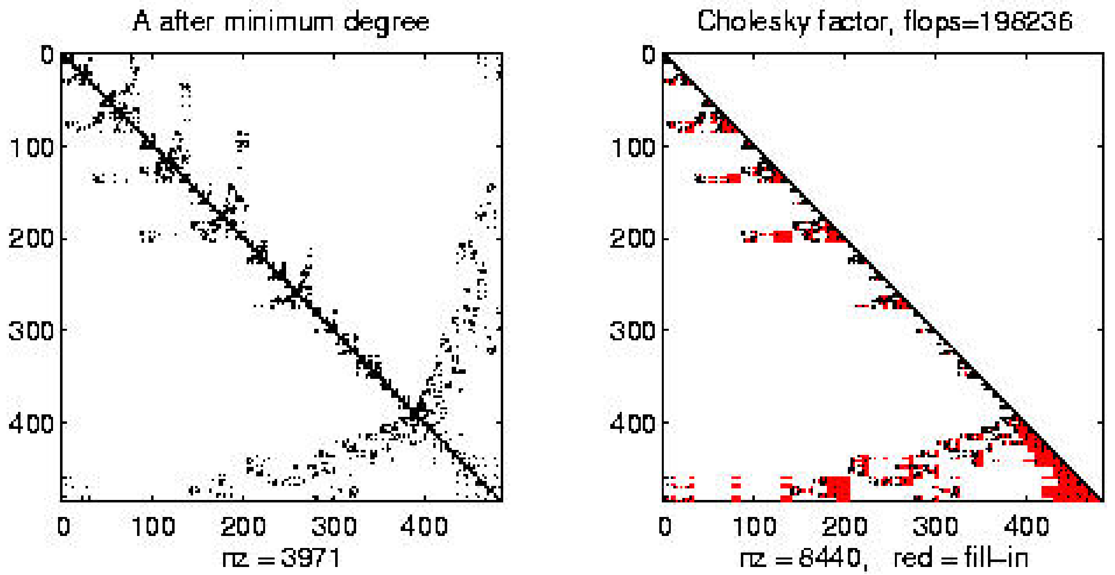


Example of Matrix Reordering Application

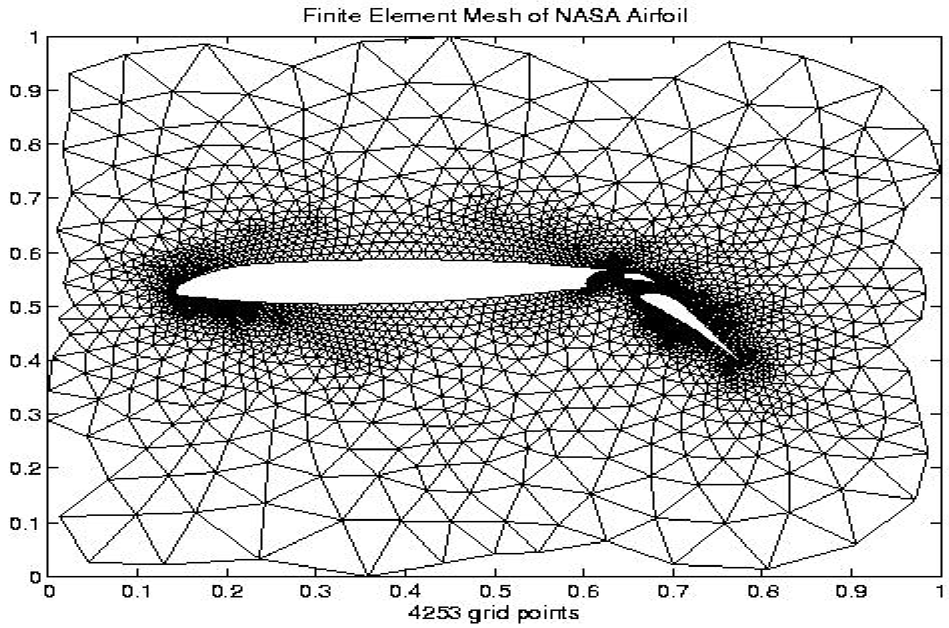
When performing
Gaussian Elimination
Zeros can be filled ☹

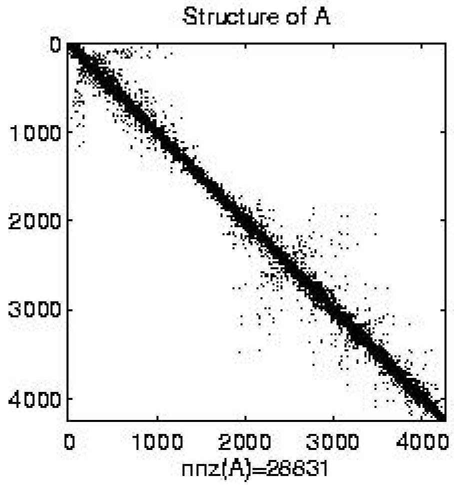


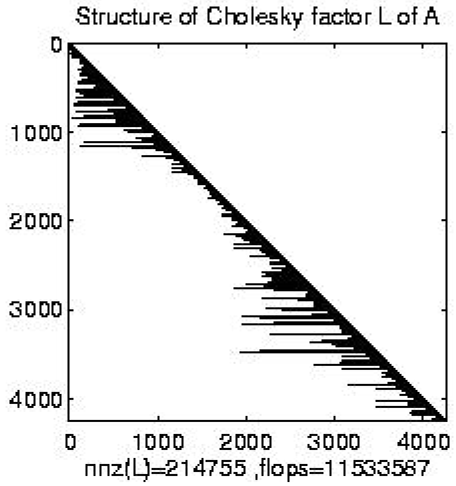
Matrix can be reordered
to reduce this fill
But it's not the same
ordering as for
parallelism



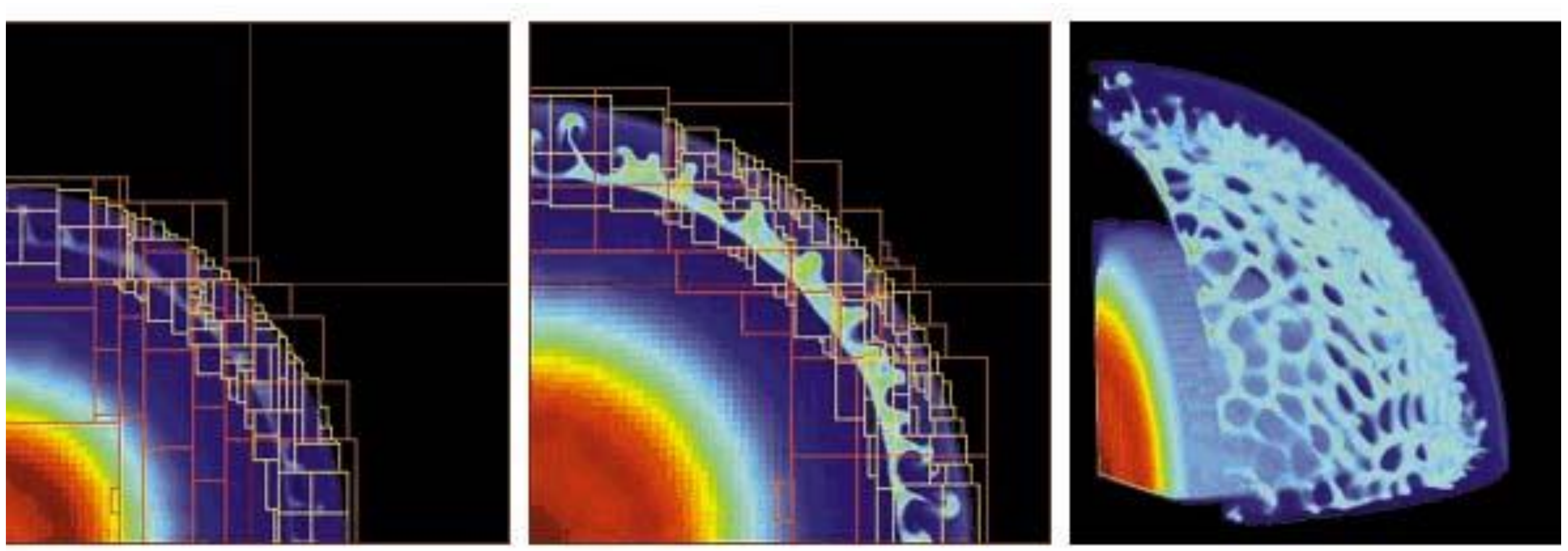
Irregular mesh: NASA Airfoil in 2D (direct solution)







Adaptive Mesh Refinement (AMR)



- Adaptive mesh around an explosion
  - Refinement done by estimating errors; refine mesh if too large
- Parallelism
  - Mostly between "patches" assigned to processors for load balance
  - May exploit parallelism within a patch

Challenges of Irregular Meshes

- How to generate them in the first place
    - Start from geometric description of object
    - 2D hard, 3D harder!
- How to partition them
    - ParMetis, a parallel graph partitioner
- How to design iterative solvers
- How to design direct solvers

- These are challenges to do sequentially, more so in parallel

Summary – sources of parallelism and locality

Attempts to categorize main "kernels" dominating simulation codes:

- Structured grids
  - including locally structured grids, as in AMR
- Unstructured grids
- Spectral methods (Fast Fourier Transform)
- Dense Linear Algebra
- Sparse Linear Algebra
  - Both explicit (SpMV) and implicit (solving)
- Particle Methods
- Monte Carlo/Embarrassing Parallelism(easy!)

Reminders

- This week's exercises: More OpenMP practice

- Next week: Start MPI distributed memory programming
    - Message passing basics and collective communication