CS 240A:

Parallelism in Physical Simulation

Partly based on slides from David Culler, Jim Demmel, Kathy Yelick, et al., UCB CS267

Parallelism and Locality in Simulation

- Real world problems have parallelism and locality:
  - Some objects may operate independently of others.
  - Objects may depend more on nearby than distant objects.
  - Dependence on distant objects can often be simplified.
- Scientific models may introduce more parallelism:
  - When a continuous problem is discretized, time-domain dependencies are generally limited to adjacent time steps.
  - Far-field effects can sometimes be ignored or approximated.
- Many problems exhibit parallelism at multiple levels
  - Example: circuits can be simulated at many levels, and within each there may be parallelism within and between subcircuits.

Multilevel Modeling: Circuit Simulation

- Circuits are simulated at many different levels

Level	Primitives	Examples
Instruction level	Instructions	SimOS, SPIM
Cycle level	Functional units	VIRAM-p
Register Transfer Level (RTL)	Register, counter, MUX	VHDL
Gate Level	Gate, flip-flop, memory cell	Thor
Switch level	Ideal transistor	Cosmos
Circuit level	Resistors, capacitors, etc.	Spice
Device level	Electrons, silicon	

Basic kinds of simulation

discrete

- Discrete event systems
  - Time and space are discrete
- Particle systems
  - Important special case of lumped systems
- Ordinary Differential Equations (ODEs)
  - Lumped systems
  - Location/entities are discrete, time is continuous
- Partial Different Equations (PDEs)
  - Time and space are continuous

continuous

Basic Kinds of Simulation

- Discrete event systems:
  - Examples: “Game of Life,” logic level circuit simulation.
- Particle systems:
  - Examples: billiard balls, semiconductor device simulation, galaxies.
- Lumped variables depending on continuous parameters:
  - ODEs, e.g., circuit simulation (Spice), structural mechanics, chemical kinetics.
- Continuous variables depending on continuous parameters:
  - PDEs, e.g., heat, elasticity, electrostatics.

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

A Model Problem: Sharks and Fish

- Illustration of parallel programming
  - Original version: WATOR, proposed by Geoffrey Fox
  - Sharks and fish living in a 2D toroidal ocean
- Several variations to show different physical phenomena
- Basic idea: sharks and fish living in an ocean
  - rules for movement
  - breeding, eating, and death
  - forces in the ocean
  - forces between sea creatures
- See link on course home page for details

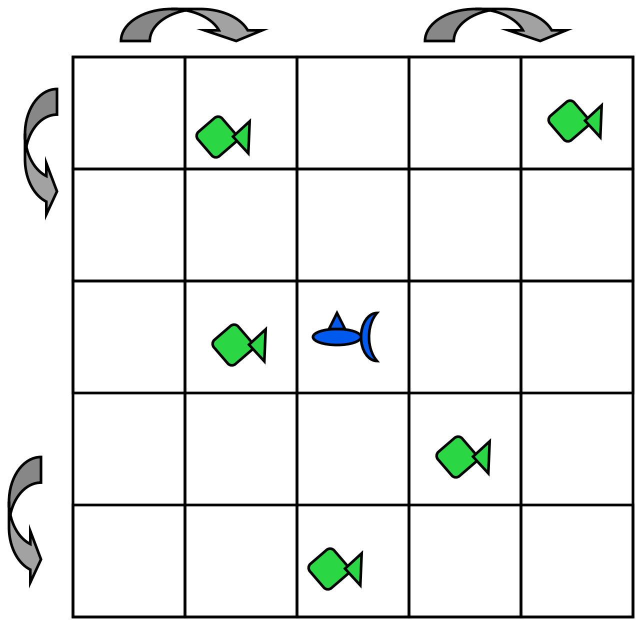
Discrete Event Systems

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 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:"
  - Also known as Sharks and Fish #3:
  - Space divided into cells, rules govern cell contents at each step

Sharks and Fish as Discrete Event System

- Ocean modeled as a 2D toroidal grid
- Each cell occupied by at most one sea creature



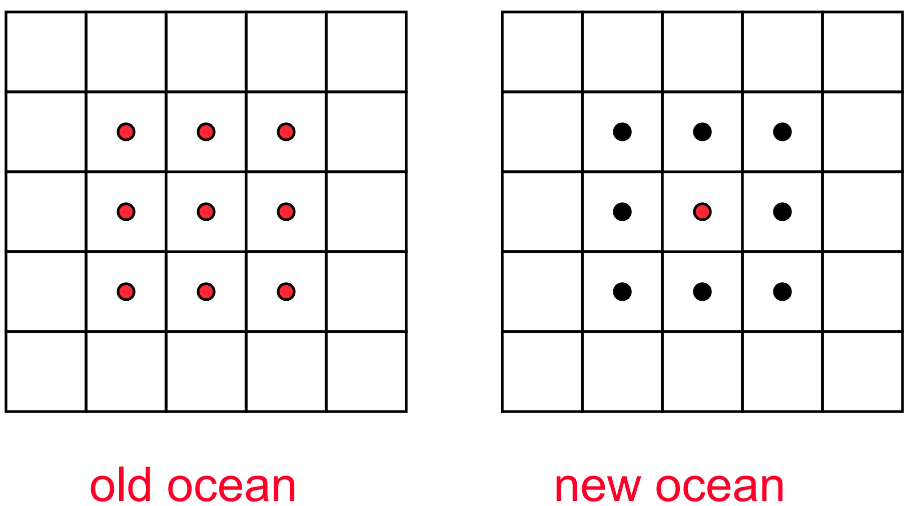
Fish-only: the Game of Life

- An new fish is born if
  - a cell is empty
  - exactly 3 (of 8) neighbors contain fish
- A fish dies (of overcrowding) if
  - cell contains a fish
  - 4 or more neighboring cells are full
- A fish dies (of loneliness) if
  - cell contains a fish
  - less than 2 neighboring cells are full
- Other configurations are stable

- The original Wator problem adds sharks that eat fish

Parallelism in Sharks and Fish

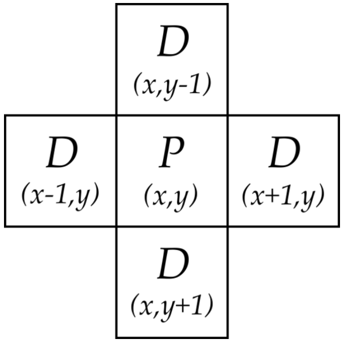
- The activities in this system are discrete events
- The simulation is synchronous
  - use two copies of the grid (old and new)
  - the value of each new grid cell in new depends only on the 9 cells (itself plus neighbors) in old grid ("stencil computation")
    - Each grid cell update is independent: reordering or parallelism OK
  - simulation proceeds in timesteps, where (logically) each cell is evaluated at every timestep



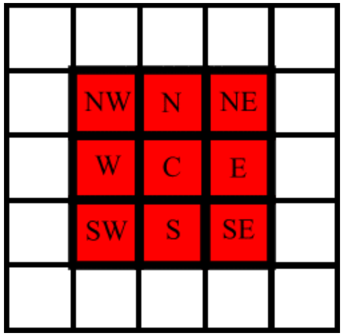
Stencil computations

- Data lives at the vertices of a regular mesh
- At each step, new values are computed from neighbors
- Examples:
    - Game of Life   (9-point stencil)
    - Matvec in 2D model problem (5-point stencil)
    - Matvec in 3D model problem (7-point stencil)

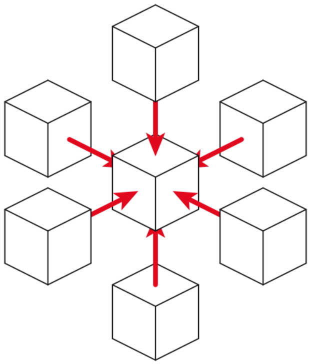
Examples of stencils



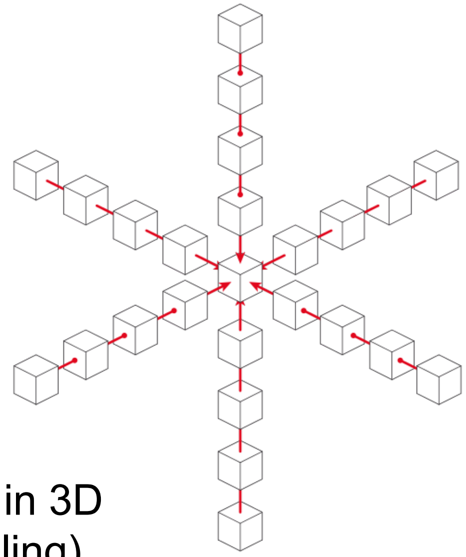
5-point stencil in 2D
(temperature problem)



9-point stencil in 2D
(game of Life)



7-point stencil in 3D
(3D temperature problem)

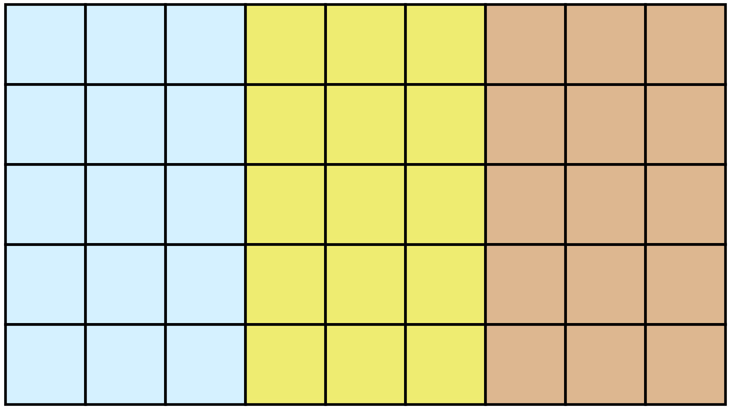


25-point stencil in 3D
(seismic modeling)

… and many more

Parallelizing Stencil Computations

- Parallelism is simple
  - Span t_∞ = constant, so potential parallelism pp = size of problem!
  - Even decomposition across processors gives load balance
- Communication volume
  - v = total # of boundary cells between patches



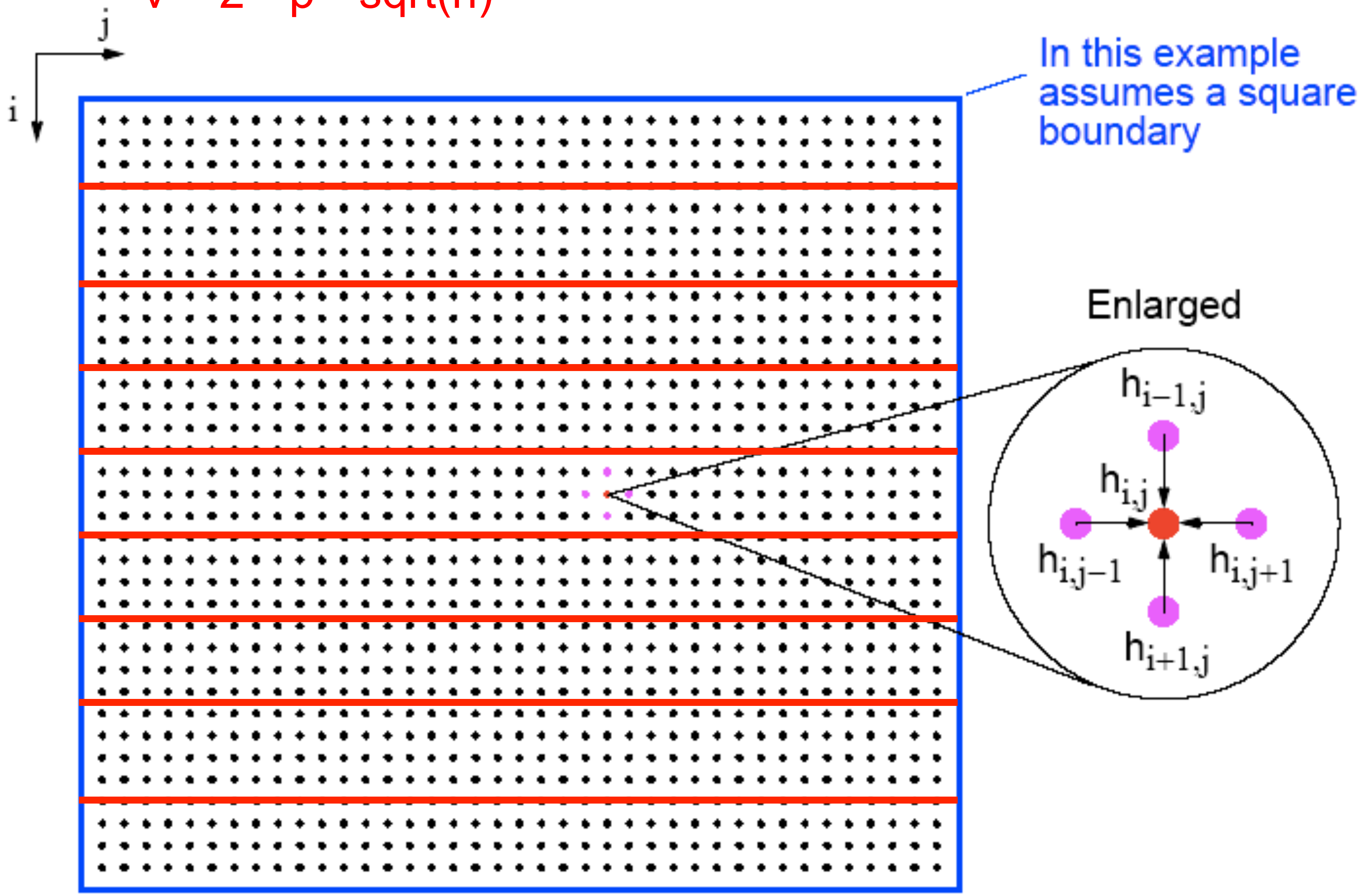
- Spatial locality limits communication cost
  - Communicate only boundary values from neighboring patches

Where's the data (5-point stencil problem)?

- Each of n stencil points has some fixed amount of data
- Divide stencil points among processors, n/p points each
- How do you divide up a sqrt(n) by sqrt(n) region of points?
- Block row (or block col) layout: v = 2 * p * sqrt(n)
- 2-dimensional block layout: v = 4 * sqrt(p) * sqrt(n)

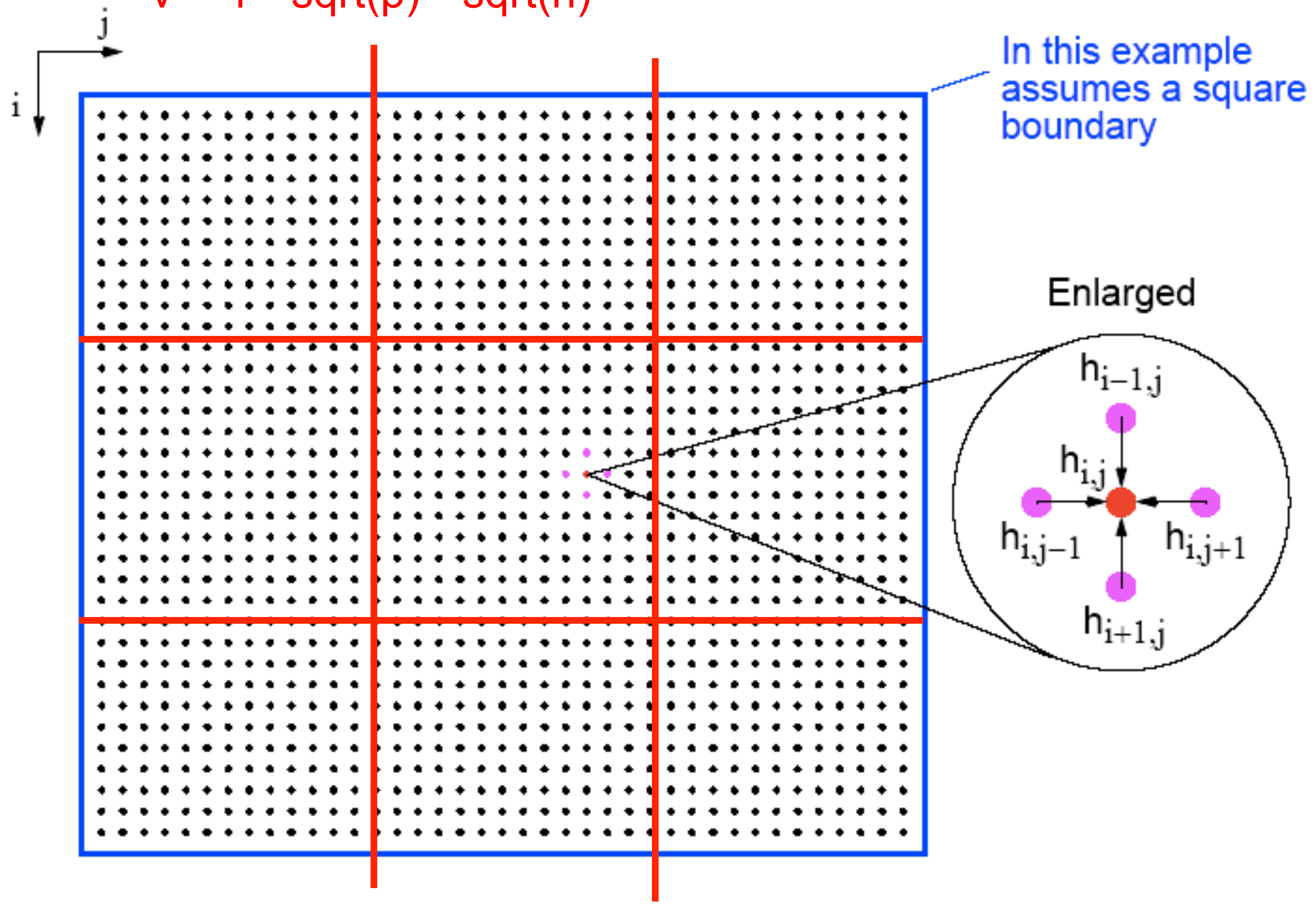
How do you partition the sqrt(n) by sqrt(n) stencil points?

- First version: number the grid by rows
- Leads to a block row decomposition of the region
- v = 2 * p * sqrt(n)



How do you partition the sqrt(n) by sqrt(n) stencil points?

- Second version: 2D block decomposition
- Numbering is a little more complicated
- v = 4 * sqrt(p) * sqrt(n)



Where's the data (temperature problem)?

- The matrix A: Nowhere!!
- The vectors x, b, r, d:
  - Each vector is one value per stencil point
  - Divide stencil points among processors, n/p points each
- How do you divide up the sqrt(n) by sqrt(n) region of points?
- Block row (or block col) layout: v = 2 * p * sqrt(n)
- 2-dimensional block layout: v = 4 * sqrt(p) * sqrt(n)

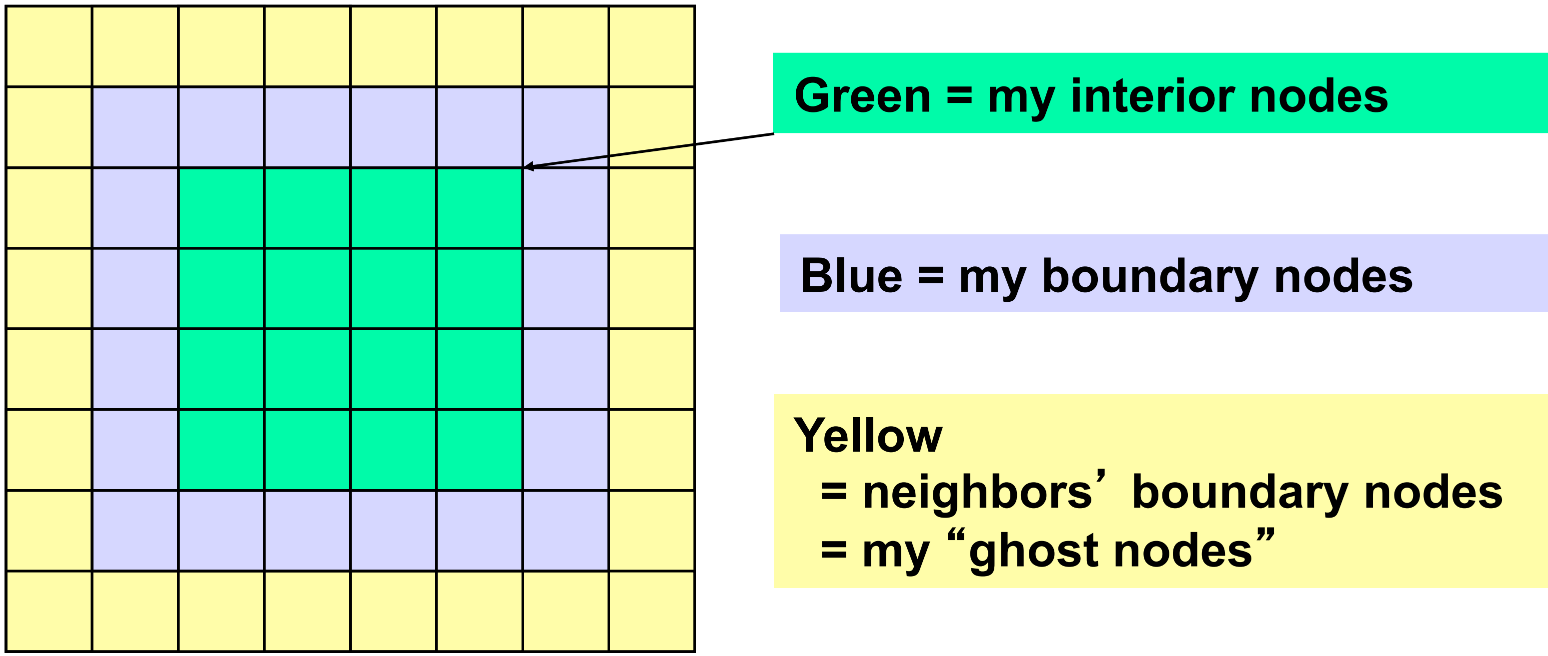
Detailed complexity measures for data movement I: Latency/Bandwidth Model

Moving data between processors by message-passing

- Machine parameters:
  - α latency (message startup time in seconds)
  - β inverse bandwidth (in seconds per word)
  - between nodes of Triton, $\alpha \sim 2.2 \times 10^{-6}$ and $\beta \sim 6.4 \times 10^{-9}$
- Time to send & recv or bcast a message of w words: $\alpha + w^*\beta$
- t_{comm} total commmunication time
- t_{comp} total computation time
- Total parallel time: $t_p = t_{comp} + t_{comm}$

Ghost Nodes in Stencil Computations

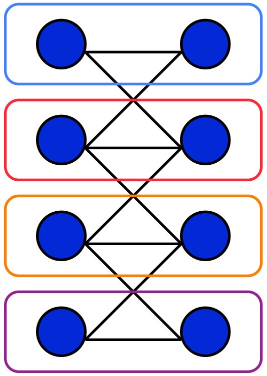
Comm cost = α * (#messages) + β * (total size of messages)



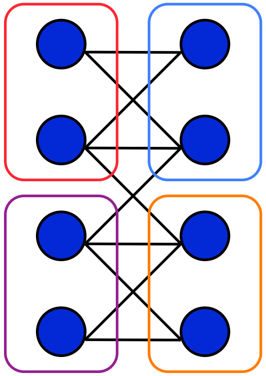
- Keep a ghost copy of neighbors' boundary nodes
- Communicate every second iteration, not every iteration
- Reduces #messages, not total size of messages
- Costs extra memory and computation
- Can also use more than one layer of ghost nodes

Synchronous Circuit Simulation

- Circuit is a graph made up of subcircuits connected by wires
  - Component simulations need to interact if they share a wire.
  - Data structure is irregular (graph) of subcircuits.
  - Parallel algorithm is timing-driven or synchronous:
    - Evaluate all components at every timestep (determined by known circuit delay)
- Graph partitioning assigns subgraphs to processors (NP-complete)
  - Determines parallelism and locality.
  - Attempts to evenly distribute subgraphs to nodes (load balance).
  - Attempts to minimize edge crossing (minimize communication).



edge crossings = 6



edge crossings = 10

Asynchronous Simulation

- Synchronous simulations may waste time:
  - Simulate even when the inputs do not change.
- Asynchronous simulations update only when an event arrives from another component:
  - No global time steps, but individual events contain time stamp.
  - Example: Game of life in loosely connected ponds (don't simulate empty ponds).
  - Example: Circuit simulation with delays (events are gates flipping).
  - Example: Traffic simulation (events are cars changing lanes, etc.).
- Asynchronous is more efficient, but harder to parallelize
  - In MPI, events can be messages …
  - … but how do you know when to "receive"?

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: laws of gravity.
  - atoms in a molecule: electrostatic forces.
  - neutrons in a fission reactor.
  - electron beam and ion beam semiconductor manufacturing.
  - cars on a freeway: Newton's laws + models of driver & engine.

- Many simulations combine particle simulation techniques with some discrete event techniques.

Forces in Particle Systems

- Force on each particle decomposed into near and far:

force = external_force + nearby_force + far_field_force

- External force
  - ocean current to sharks and fish world (S&F 1).
  - externally imposed electric field in electron beam.

- Nearby force
  - sharks attracted to eat nearby fish (S&F 5).
  - balls on a billiard table bounce off of each other.
  - Van der Waals forces in fluid ($1/r^6$).

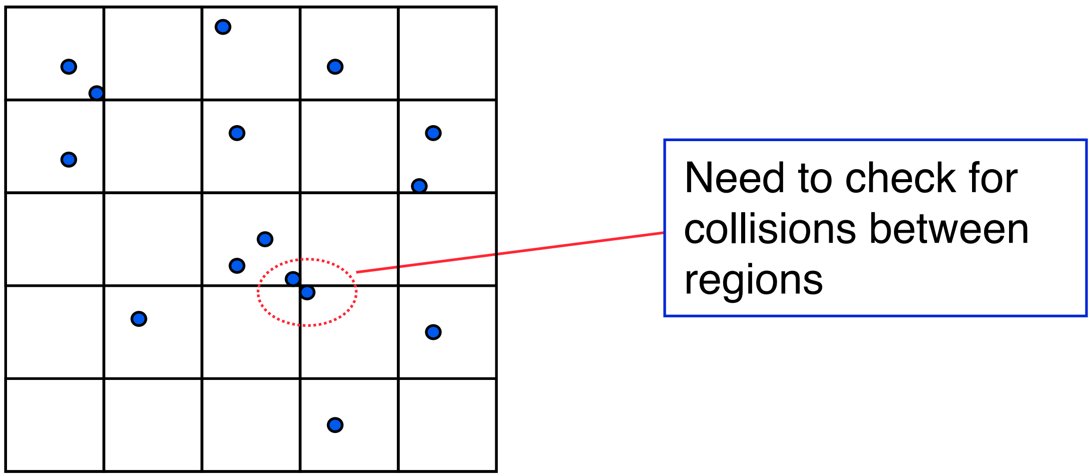
- Far-field force
  - fish attract other fish by gravity-like ($1/r^2$) force (S&F 2).
  - gravity, electrostatics
  - forces governed by elliptic PDEs.

Parallelism in External Forces

- External forces are the simplest to implement.
  - Force on one particle is independent of other particles.
  - “Embarrassingly parallel”.
- Evenly distribute particles on processors
  - Any even distribution works.
  - Locality is not an issue, since no communication.
- For each particle on processor, apply external force.

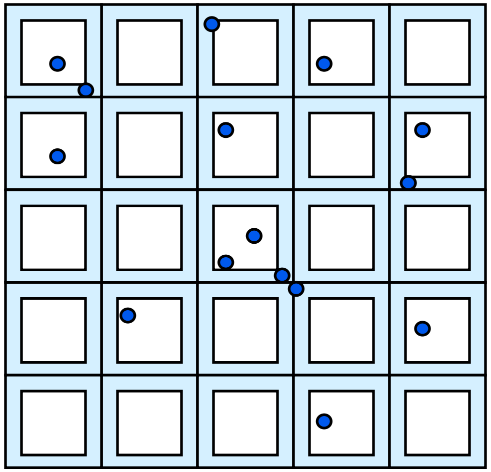
Parallelism in Nearby Forces

- Nearby forces require interaction => communication.
- Force depends on other particles nearby (e.g. collisions)
- Simple algorithm: check every pair for collision: $O(n^2)$
- Parallelism by decomposition of physical domain:
  - $O(n/p)$ particles per processor if evenly distributed.
- Better algorithm: only check pairs near boundaries



Parallelism in Nearby Forces

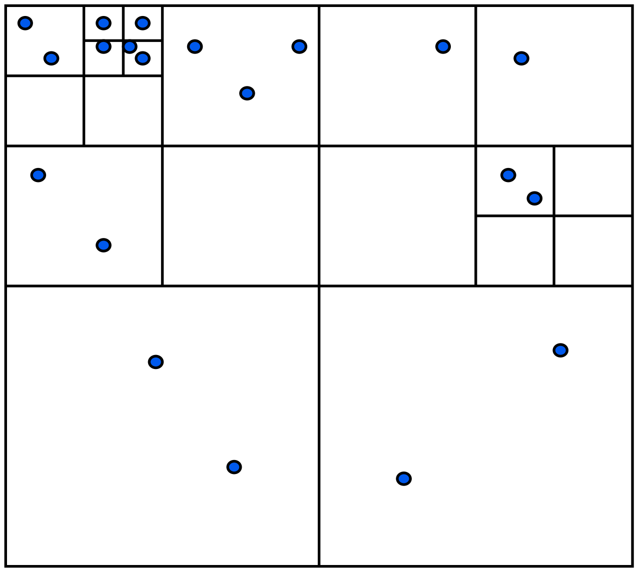
- Challenge 1: interactions of particles near boundaries:
  - Communicate particles near boundary to neighboring processors.
  - Surface to volume effect limits communication.
  - Which communicates less: squares (as below) or slabs?



Communicate particles in boundary region to neighbors

Parallelism in Nearby Forces

- Challenge 2: load imbalance, if particles cluster together:
  - Stars in galaxies, for example
- To reduce load imbalance, divide space unevenly.
  - Each region contains roughly equal number of particles.
  - Quad-tree in 2D, oct-tree in 3D.



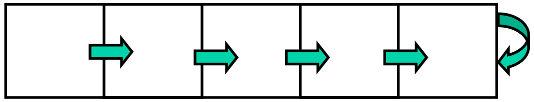
Example: each square contains at most 3 particles

See: http://njord.umiacs.umd.edu:1601/users/brabec/quadtree/points/prquad.html

Parallelism in Far-Field Forces

- Far-field forces involve all-to-all interaction and communication.
- Force on one particle depends on all other particles.

  - Examples: galaxies (gravity), protein folding (electrostatics)
  - Simplest algorithm is $O(n^2)$ as in S&F 2, 4, 5.
  - Decomposing space does not help total work or communication, since every particle needs to "visit" every other particle.



Implement by rotating particle sets.

- Keeps processors busy
- All processors see all particles
- Just like MGR matrix multiply!

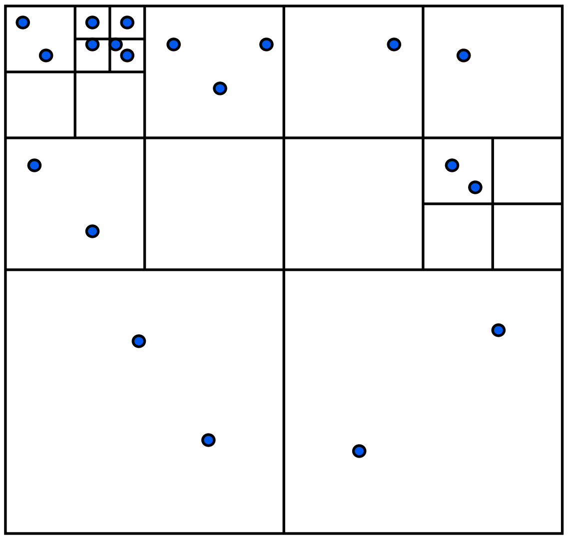
Use more clever algorithms to beat $O(n^2)$?

Far-field forces: Tree Decomposition

- "Fast multipole" algorithms
  - Approximate the force from far-away particles
  - Simplify a group of far-away particles into a single multipole.
  - Do this at <u>every</u> scale simultaneously (every quadtree level)
  - Each quadtree node contains an approximation of descendants.

- O(n log n) or even O(n) instead of $O(n^2)$.

- "Top 10 Algorithms of the 20th Century" (resources page)
- Tutorial on course web page.



Summary of Particle Methods

- Model contains discrete entities, namely, particles

  force = external_force + nearby_force + far_field_force

- Time is continuous – is discretized to solve

- Simulation follows particles through timesteps

  - All-pairs algorithm is simple, but inefficient, $O(n^2)$
  - Particle-mesh methods approximates by moving particles
  - Tree-based algorithms approximate by treating set of particles as a group, when far away

- This is a special case of a “lumped” system . . .

Lumped Systems: ODEs

System of Lumped Variables

- Finitely many variables
- Depending on a continuous parameter (usually time)

- Example 1 – System of chemical reactions:
  - Each reaction consumes some "compounds" and produces others
  - Stoichiometric matrix S: rows for compounds, cols for reactions
  - Compound concentrations x(i) in terms of reaction rates v(j):

$$dx/dt = S * v$$

- Example 2 – Electronic circuit:
  - Circuit is a graph.
    - wires are edges.
    - each edge has resistor, capacitor, inductor or voltage source.
  - Variables are voltage & current at endpoints of edges.
  - Related by Ohm's Law, Kirchoff's Laws, etc.

- Forms a system of ordinary differential equations (ODEs).
  - Differentiated with respect to time

Example: Stoichiometry in chemical reactions

reaction 1:	$CP \Rightarrow PC$
reaction 2:	$C + P \Rightarrow CP$
reaction 3:	$C + AP \Rightarrow CP + A$

- Matrix S : row = compound, column = reaction
- Linear ODE system: d/dt (concentration) = S * (reaction rate)

$$
\begin{array}{l}
\text{compound A:} \\
\text{compound C:} \\
\text{compound P:} \\
\text{compound CP:} \\
\text{compound AP:} \\
\text{compound PC:}
\end{array}
\quad
\begin{pmatrix} dx_1/dt \\ dx_2/dt \\ dx_3/dt \\ dx_4/dt \\ dx_5/dt \\ dx_6/dt \end{pmatrix}
=
\begin{pmatrix} 0 & 0 & 1 \\ 0 & -1 & -1 \\ 0 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{pmatrix}
*
\begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}
$$

Example: Electronic circuit

- State of the system is represented by

$$\left.\begin{array}{l} \bullet\ v_n(t)\ \text{node voltages} \\ \bullet\ i_b(t)\ \text{branch currents} \\ \bullet\ v_b(t)\ \text{branch voltages} \end{array}\right\}\ \text{all at time } t$$

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

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

- Write as single large system of ODEs (possibly with constraints).

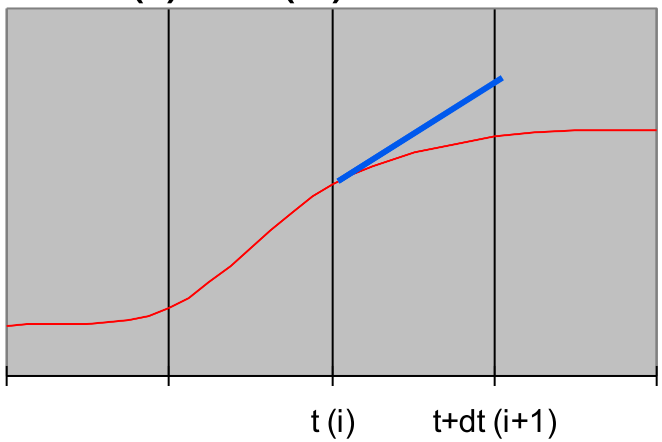
Solving ODEs

- In most examples, the matrices are sparse:
  - most array elements are 0.
  - neither store nor compute on these 0's.

- 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

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

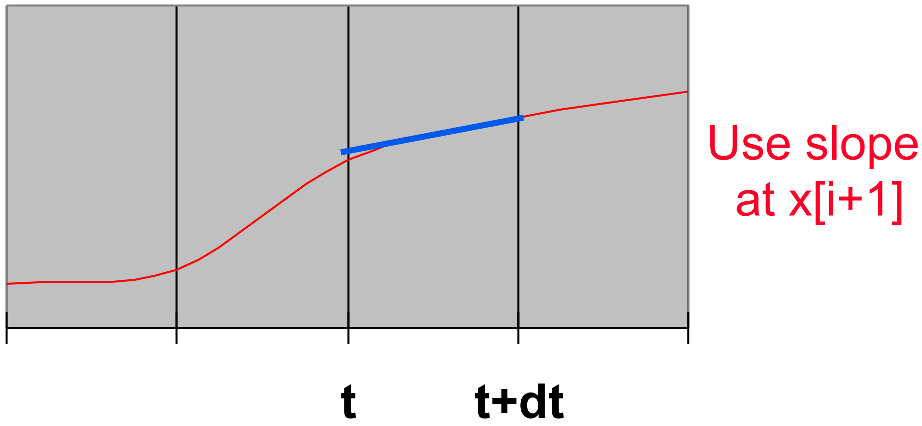


Use slope at x[i]

- Explicit methods, e.g., (Forward) Euler's method.
  - Approximate x' (t)=A*x 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.

Solving ODEs: Implicit Methods

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



- Implicit method, e.g., Backward Euler solve:
  - Approximate x'(t)=A*x 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
  - Harder algorithm: need to solve a sparse system at each step

ODEs and Sparse Matrices

- All these reduce to sparse matrix problems
  - Explicit: sparse matrix-vector multiplication.
  - Implicit: solve a sparse linear system
    - direct solvers (Gaussian elimination).
    - iterative solvers (use sparse matrix-vector multiplication).
  - Eigenvalue/eigenvector algorithms may also be either explicit or implicit.

Partial Differential Equations (PDEs)

Continuous Variables, Continuous Parameters

Examples:

- Parabolic (time-dependent) problems:
  - Heat flow: Temperature(position, time)
  - Diffusion: Concentration(position, time)
- Elliptic (steady state) problems:
  - Electrostatic or Gravitational Potential: Potential(position)
- Hyperbolic problems (waves):
  - Quantum mechanics: Wave-function(position,time)

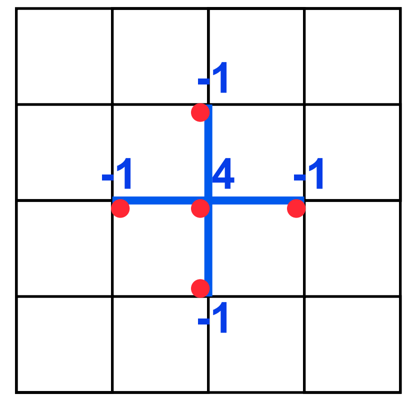
Many problems combine features of above

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

2D Implicit Method

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

Graph and "5 point stencil"

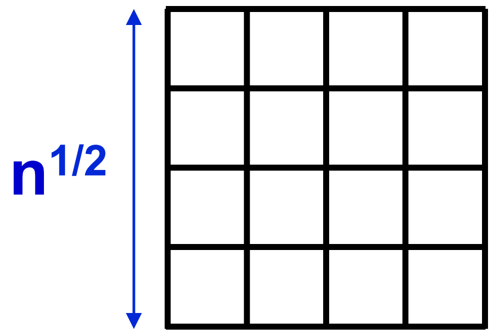


3D case is analogous
(7 point stencil)

- Multiplying by this matrix is just nearest neighbor computation on 2D grid.

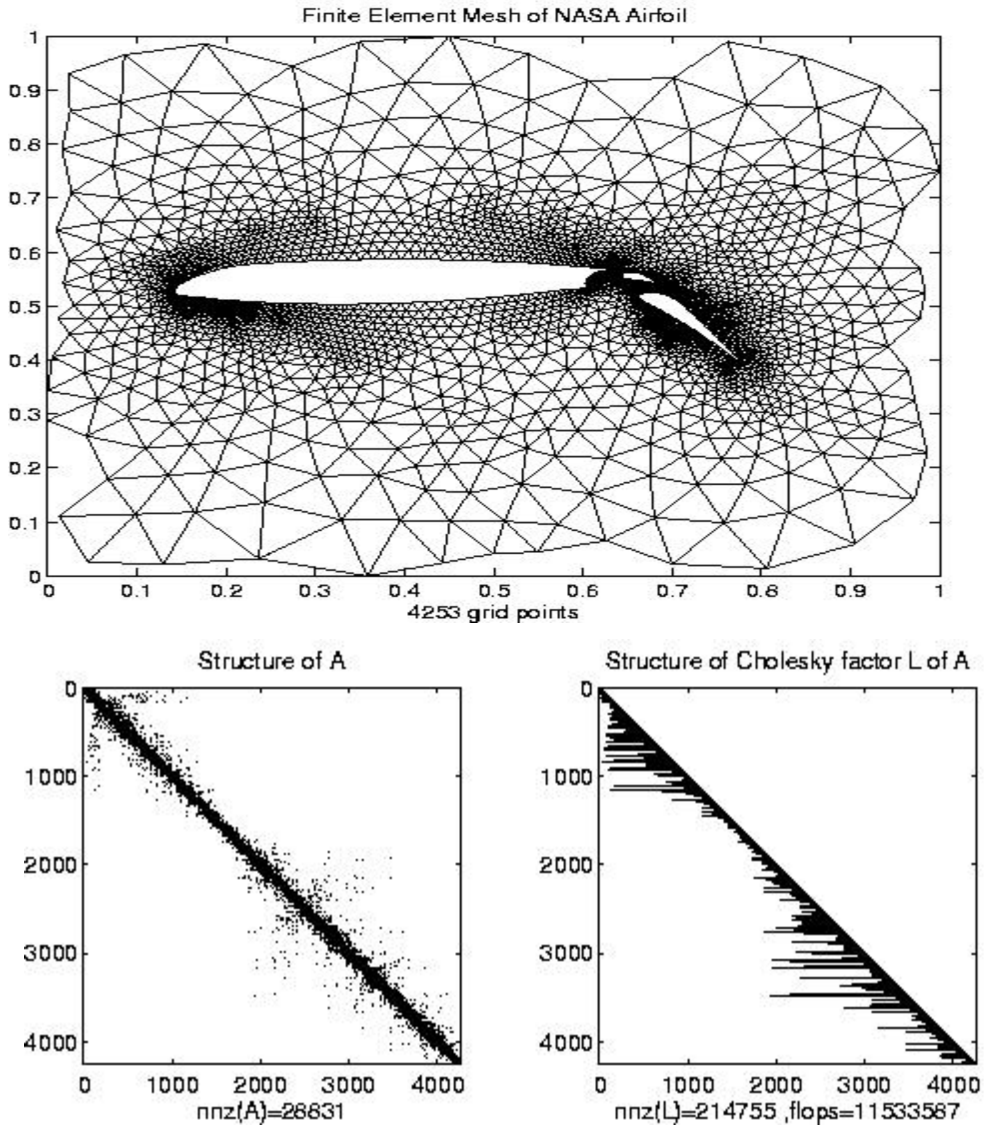
- To solve this system, there are several techniques.

The (2-dimensional) model problem

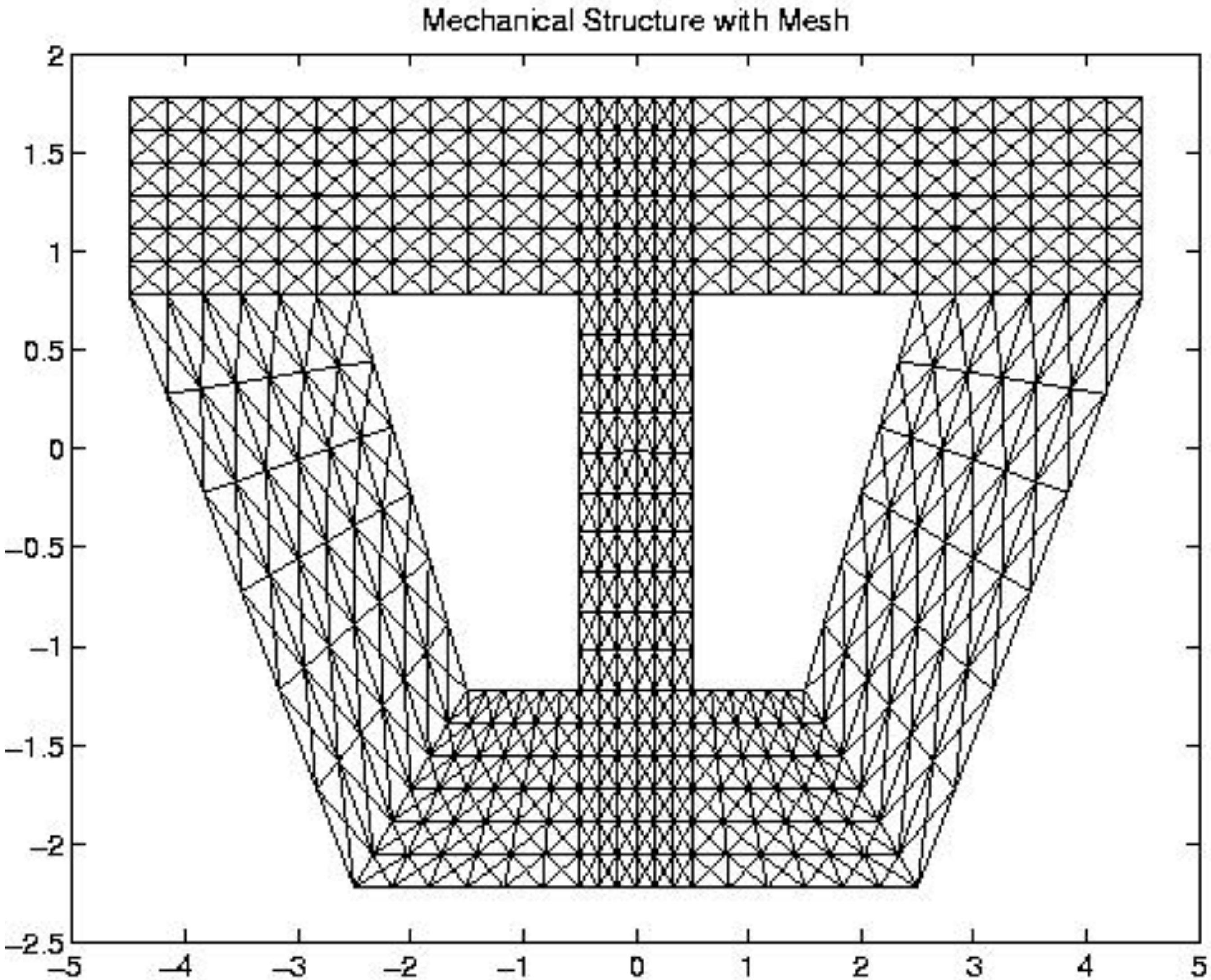


- Graph is a regular square grid with n = k^2 vertices.
- Corresponds to matrix for regular 2D finite difference mesh.
- Gives good intuition for behavior of sparse matrix algorithms on many 2-dimensional physical problems.
- There's also a 3-dimensional model problem.

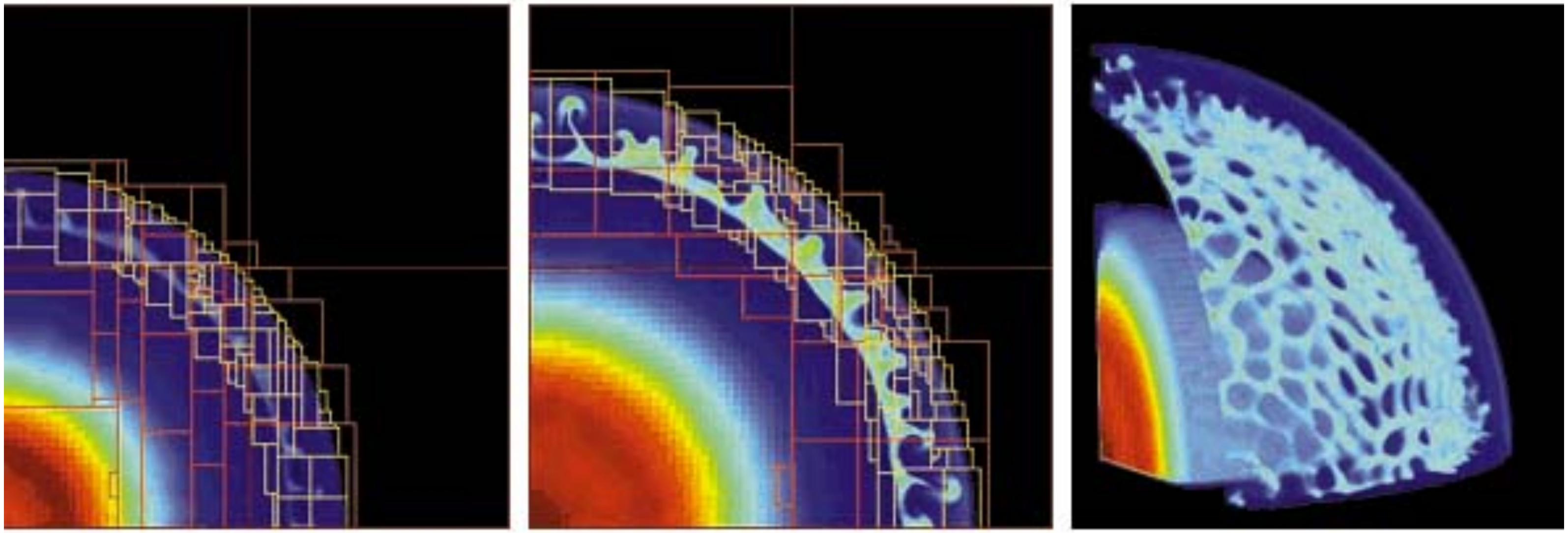
Irregular mesh: NASA Airfoil in 2D



Composite Mesh from a Mechanical Structure

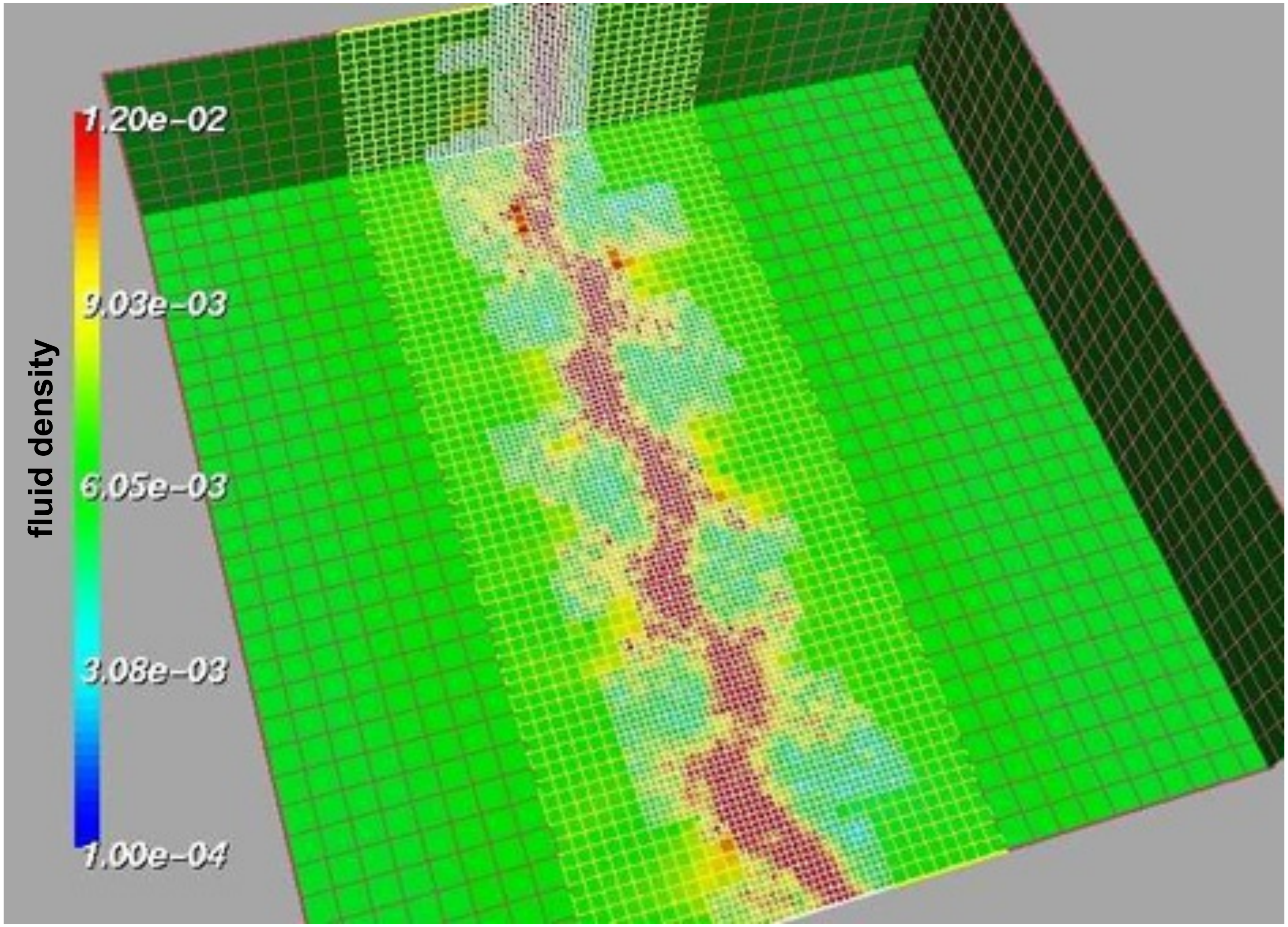


Adaptive Mesh Refinement (AMR)



- Adaptive mesh around an explosion
  - Refinement done by calculating errors
- Parallelism
  - Mostly between "patches," dealt to processors for load balance
  - May exploit some within a patch (SMP)

Adaptive Mesh



Shock waves in a gas dynamics using AMR (Adaptive Mesh Refinement) See: http://www.llnl.gov/CASC/SAMRAI/

Irregular mesh: Tapered Tube (Multigrid)

Example of Prometheus meshes

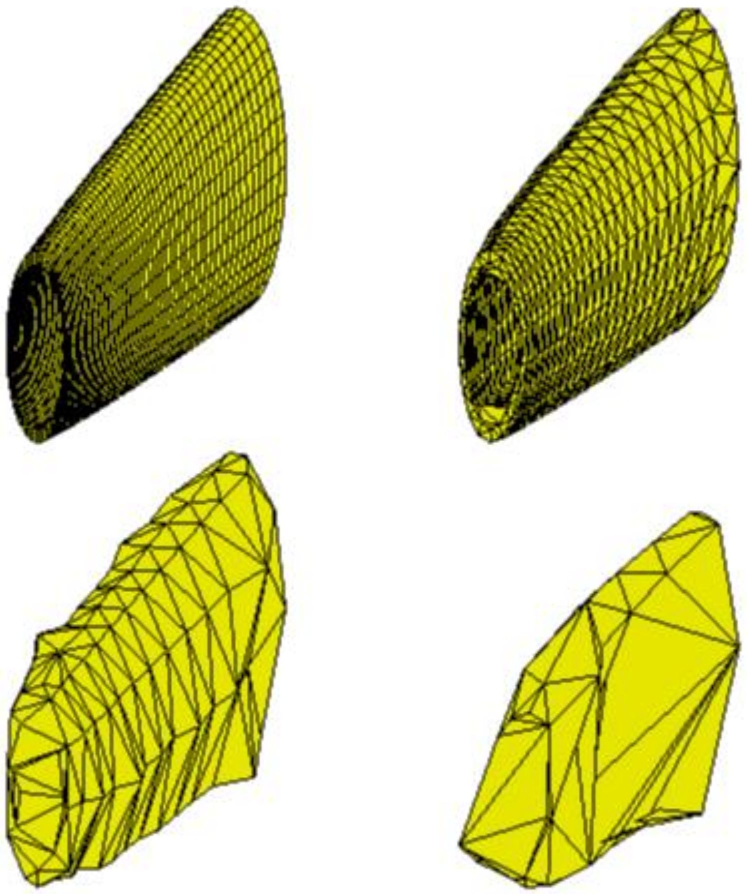
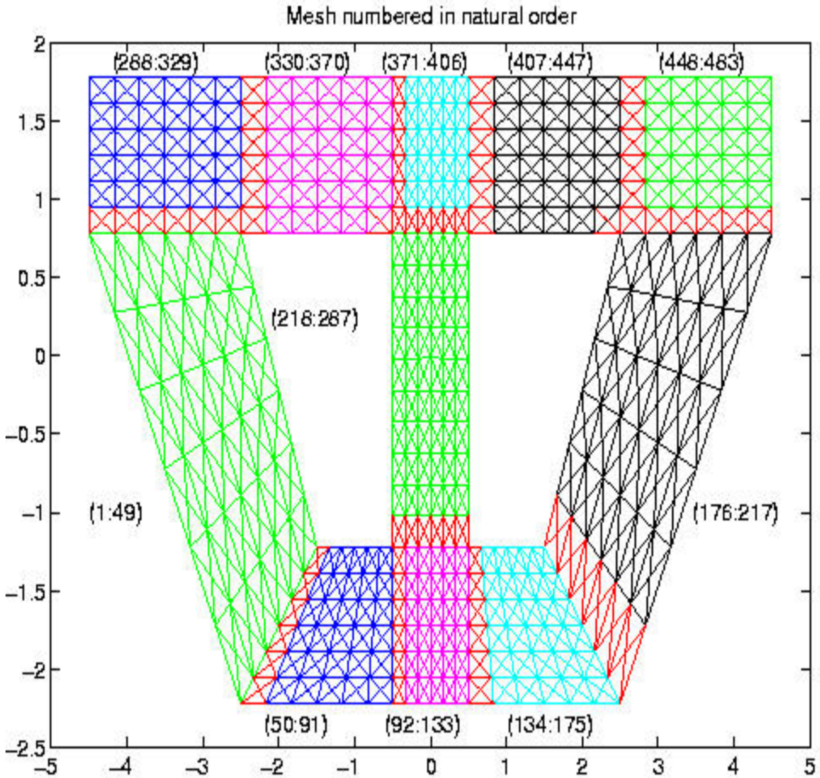
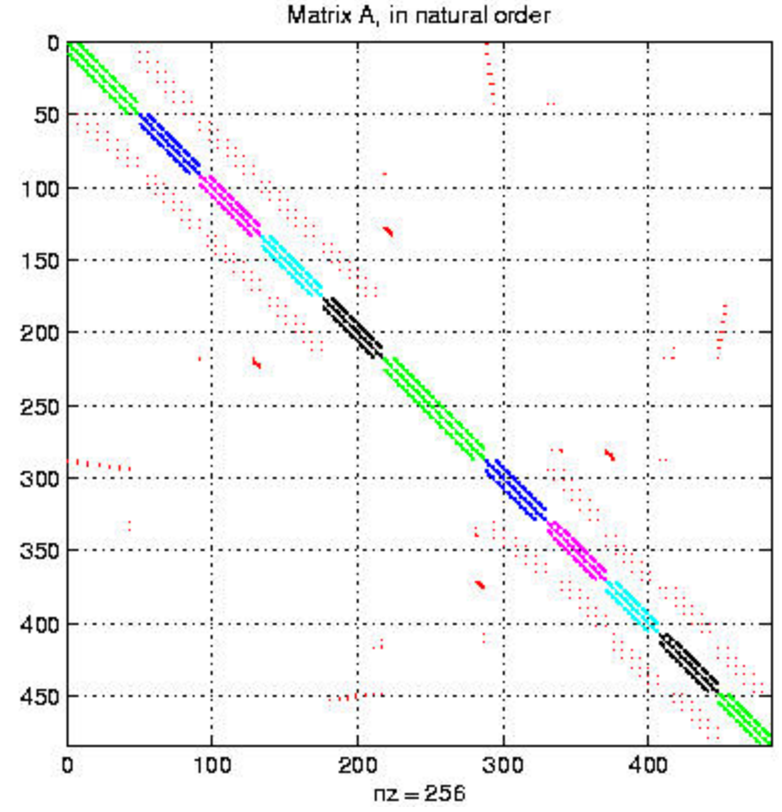


Figure 6: Sample input grid and coarse grids

Converting the Mesh to a Matrix





The Landscape of Ax = b Algorithms

	Gaussian elimination	Iterative
Any matrix	Pivoting LU	GMRES, BiCGSTAB, …
Symmetric positive definite matrix	Cholesky	Conjugate gradient

More General

More Robust

More Robust

Less Storage

Conjugate gradient in general

- CG can be used to solve *any* system $Ax = b$, if …

Conjugate gradient in general

- CG can be used to solve *any* system $Ax = b$, if …
- The matrix A is *symmetric* ($a_{ij} = a_{ji}$) …
- … and *positive definite* (all eigenvalues > 0).

Conjugate gradient in general

- CG can be used to solve *any* system $Ax = b$, if …
- The matrix A is *symmetric* ($a_{ij} = a_{ji}$) …
- … and *positive definite* (all eigenvalues > 0).

- Symmetric positive definite matrices occur a lot in scientific computing & data analysis!

Conjugate gradient in general

- CG can be used to solve *any* system $Ax = b$, if …
- The matrix A is *symmetric* ($a_{ij} = a_{ji}$) …
- … and *positive definite* (all eigenvalues > 0).

- Symmetric positive definite matrices occur a lot in scientific computing & data analysis!

- But usually the matrix isn't just a stencil.
- Now we do need to store the matrix A. Where's the data?

Conjugate gradient in general

- CG can be used to solve *any* system $Ax = b$, if …
- The matrix A is *symmetric* ($a_{ij} = a_{ji}$) …
- … and *positive definite* (all eigenvalues > 0).

- Symmetric positive definite matrices occur a lot in scientific computing & data analysis!

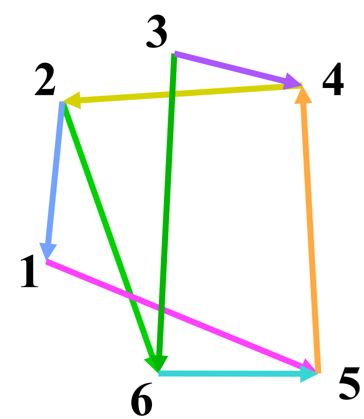
- But usually the matrix isn't just a stencil.
- Now we do need to store the matrix A. Where's the data?

- The key is to use graph data structures and algorithms.

Graphs and Sparse Matrices

- Sparse matrix is a representation of a (sparse) graph

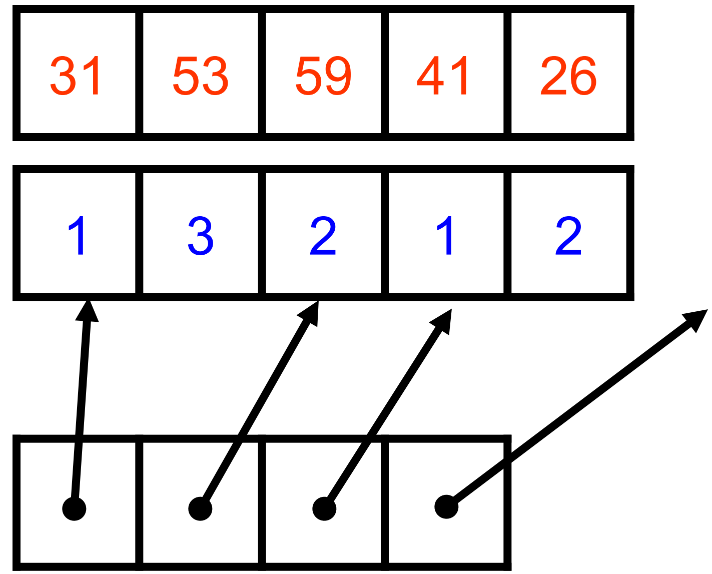
	1	2	3	4	5	6
1	1				1	
2	1	1				1
3			1	1		1
4		1		1		
5				1	1	
6					1	1



- Matrix entries are edge weights
- Number of nonzeros per row is the vertex degree
- Edges represent data dependencies in matrix-vector multiplication

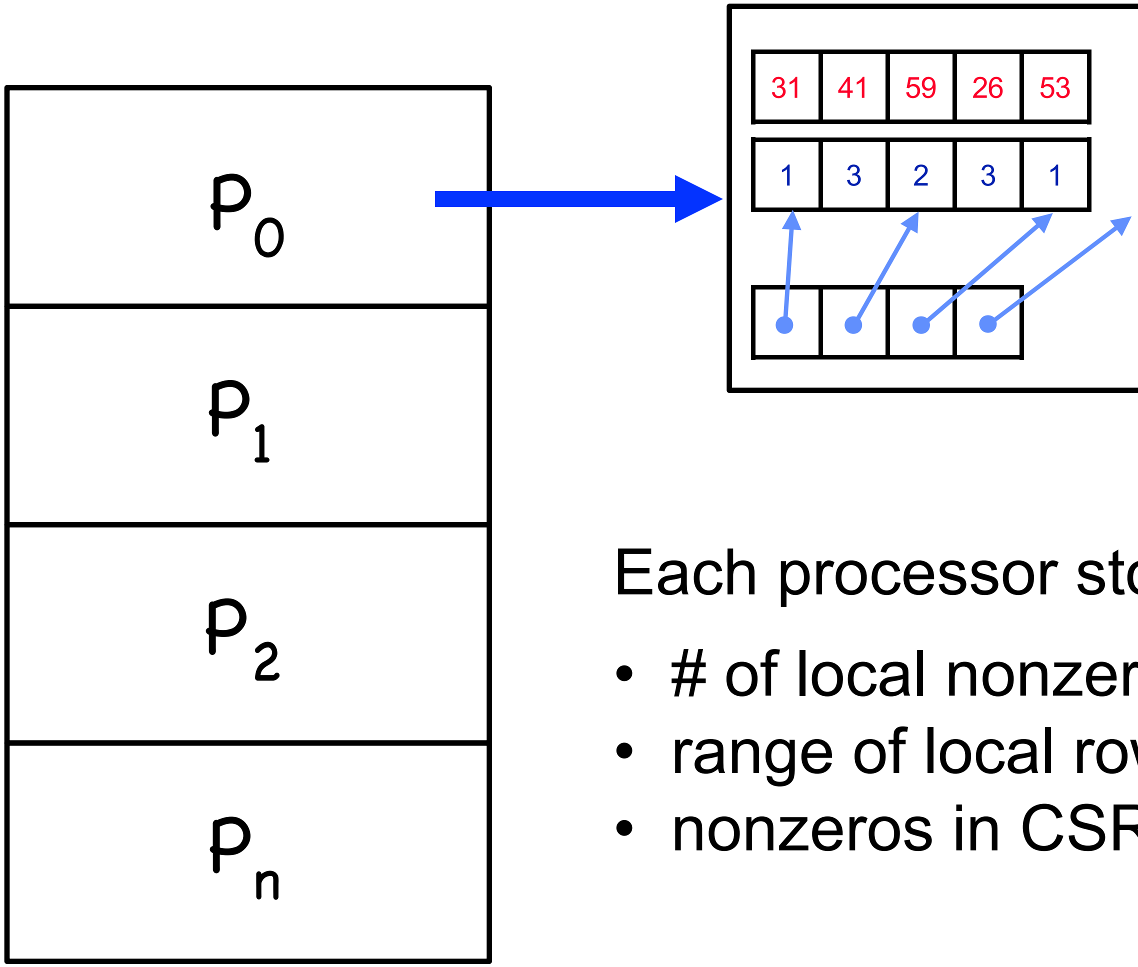
Data structure for sparse matrix A (stored by rows)

31	0	53
0	59	0
41	26	0



- Full matrix:
  - 2-dimensional array of real or complex numbers
  - (nrows*ncols) memory
- Sparse matrix:
  - compressed row storage
  - about (2*nzs + nrows) memory

Distributed-memory sparse matrix data structure



Each processor stores:

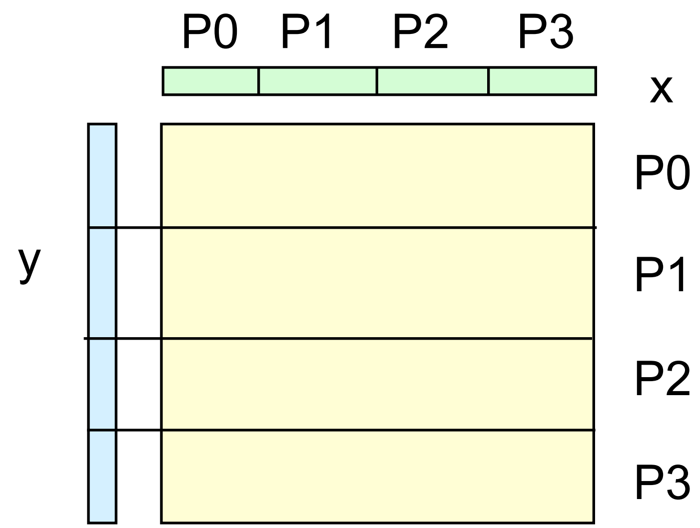
- # of local nonzeros
- range of local rows
- nonzeros in CSR form

Vector and matrix primitives for CG

- DAXPY: $v = \alpha * v + \beta * w$ (vectors v, w; scalars α, β)
  - Broadcast the scalars α and β, then independent * and +
  - comm volume = 2p, span = log n
- DDOT: $\alpha = v^T * w = \Sigma_j v[j]*w[j]$ (vectors v, w; scalar α)
  - Independent *, then + reduction
  - comm volume = p, span = log n
- Matvec: $v = A*w$ (matrix A, vectors v, w)
  - The hard part
  - But all you need is a subroutine to compute v from w
  - Sometimes you don't need to store A (e.g. temperature problem)
  - Usually you do need to store A, but it's *sparse* ...

Parallel Dense Matrix-Vector Product

- y = A*x, where A is a dense matrix



- Layout:
  - 1D by rows
- Algorithm:

  Foreach processor j

    Broadcast X(j)

    Compute A(p)*x(j)
- A(i) is the n by n/p block row that processor Pi owns
- Algorithm uses the formula

  $Y(i) = A(i)*X = \Sigma_j A(i)*X(j)$

Parallel sparse matrix-vector product

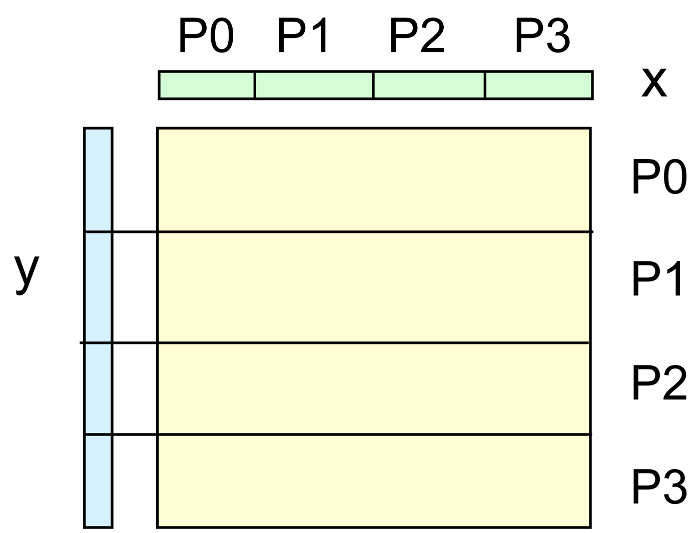
- Lay out matrix and vectors by rows
- y(i) = sum(A(i,j)*x(j))
- Only compute terms with A(i,j) ≠ 0

- <u>Algorithm</u>

  Each processor i:

    Broadcast x(i)

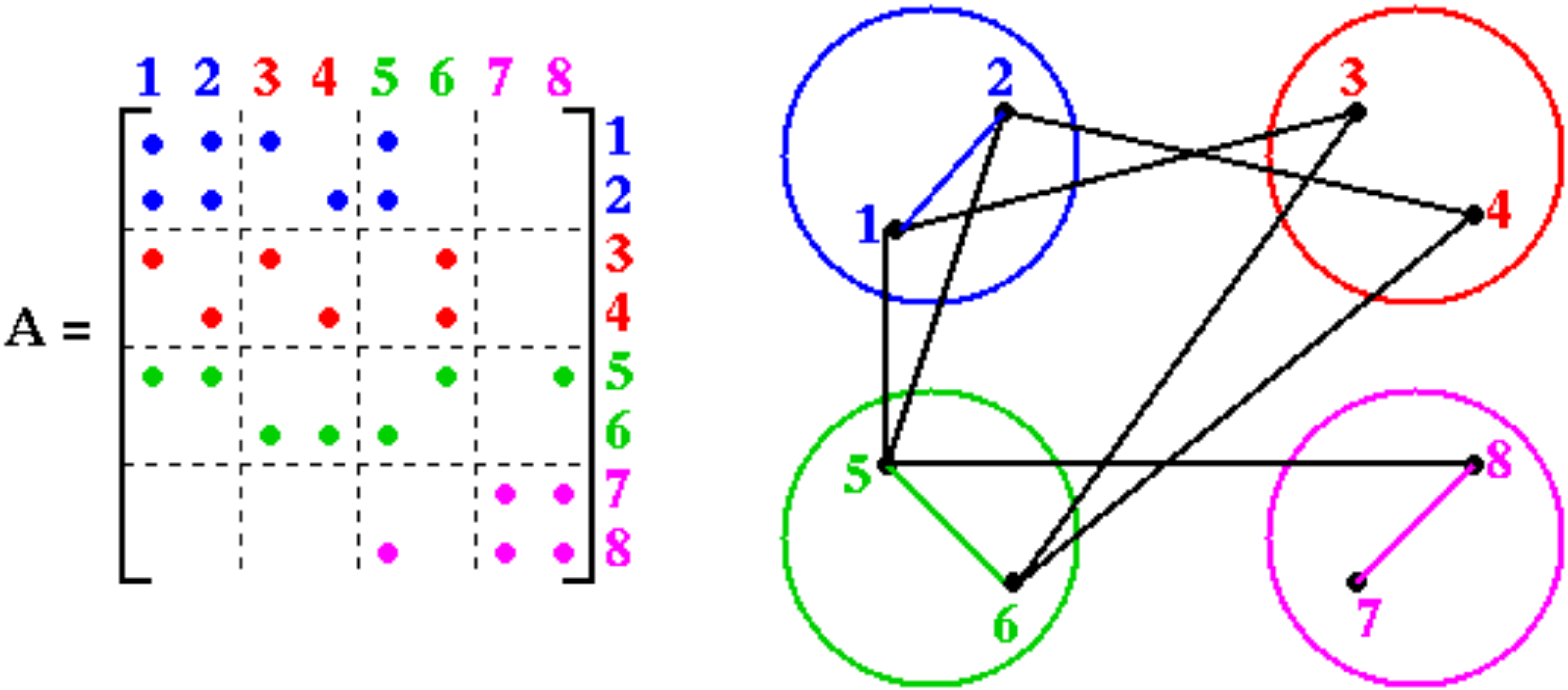
    Compute y(i) = A(i,:)*x



- <u>Optimizations</u>
  - Only send each proc the parts of x it needs, to reduce comm
  - Reorder matrix for better locality by graph partitioning
  - Worry about balancing number of nonzeros / processor, if rows have very different nonzero counts

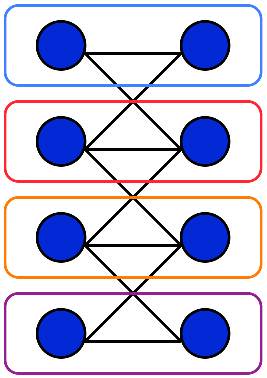
Sparse Matrix-Vector Multiplication

Partitioning a Sparse Symmetric Matrix

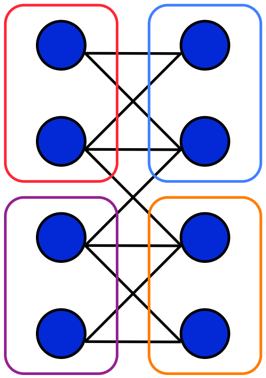


Graph partitioning (topic of later lecture)

- Assigns subgraphs to processors
- Determines parallelism and locality.
- Tries to make subgraphs all same size (load balance)
- Tries to minimize edge crossings (communication).
- Exact minimization is NP-complete.
- See Matlab demo.

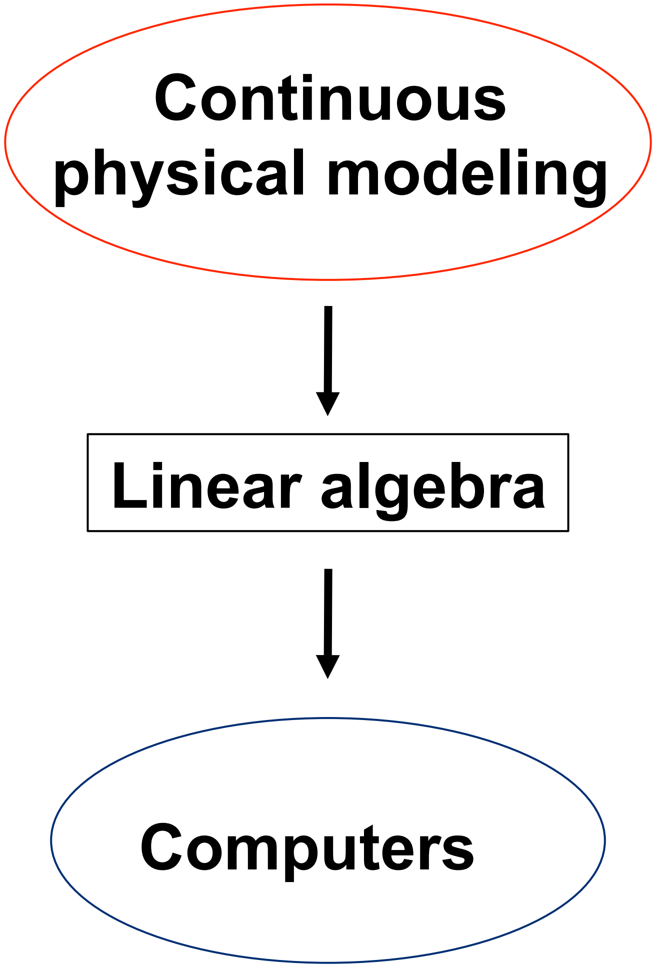


edge crossings = 6

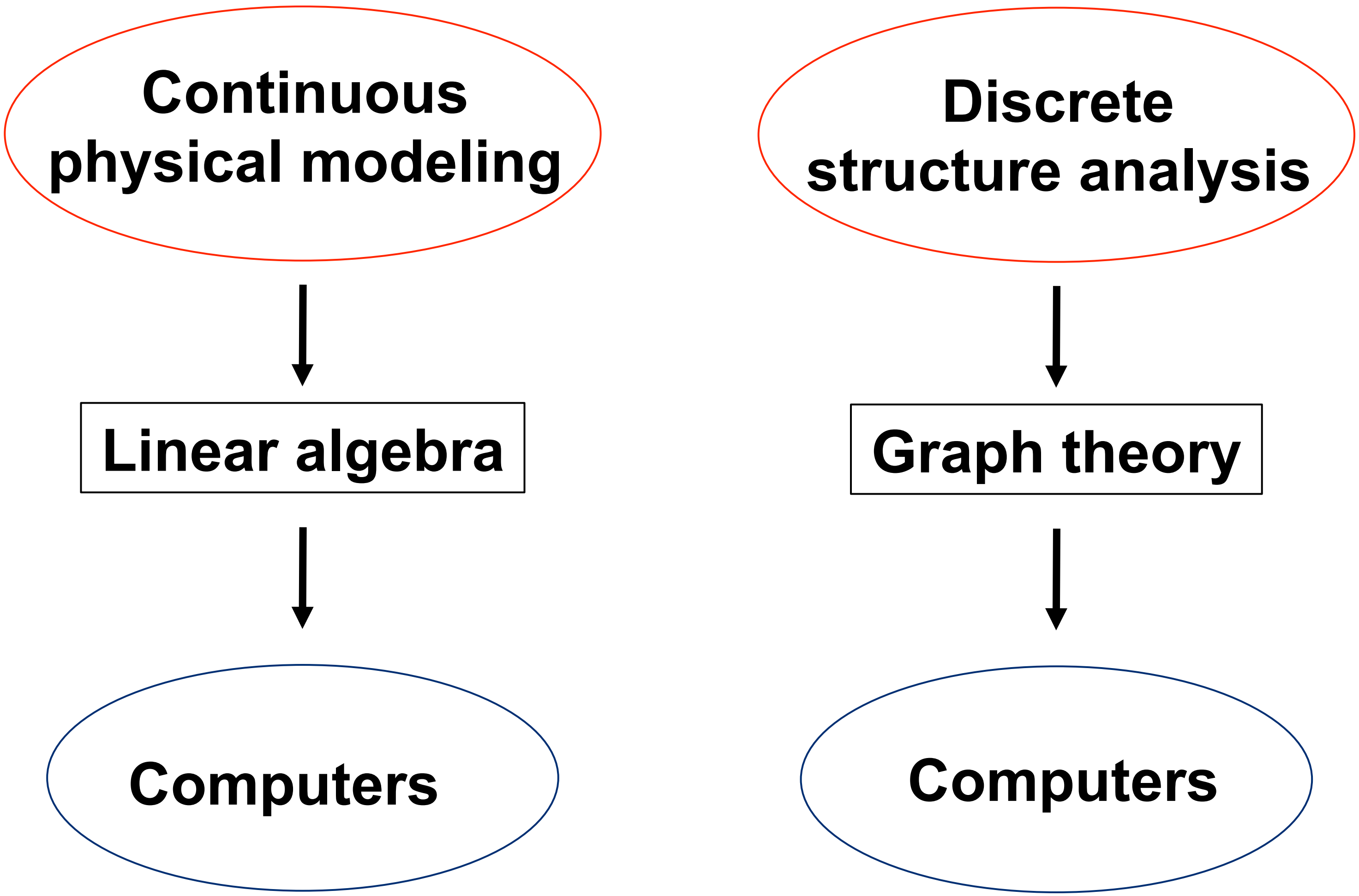


edge crossings = 10

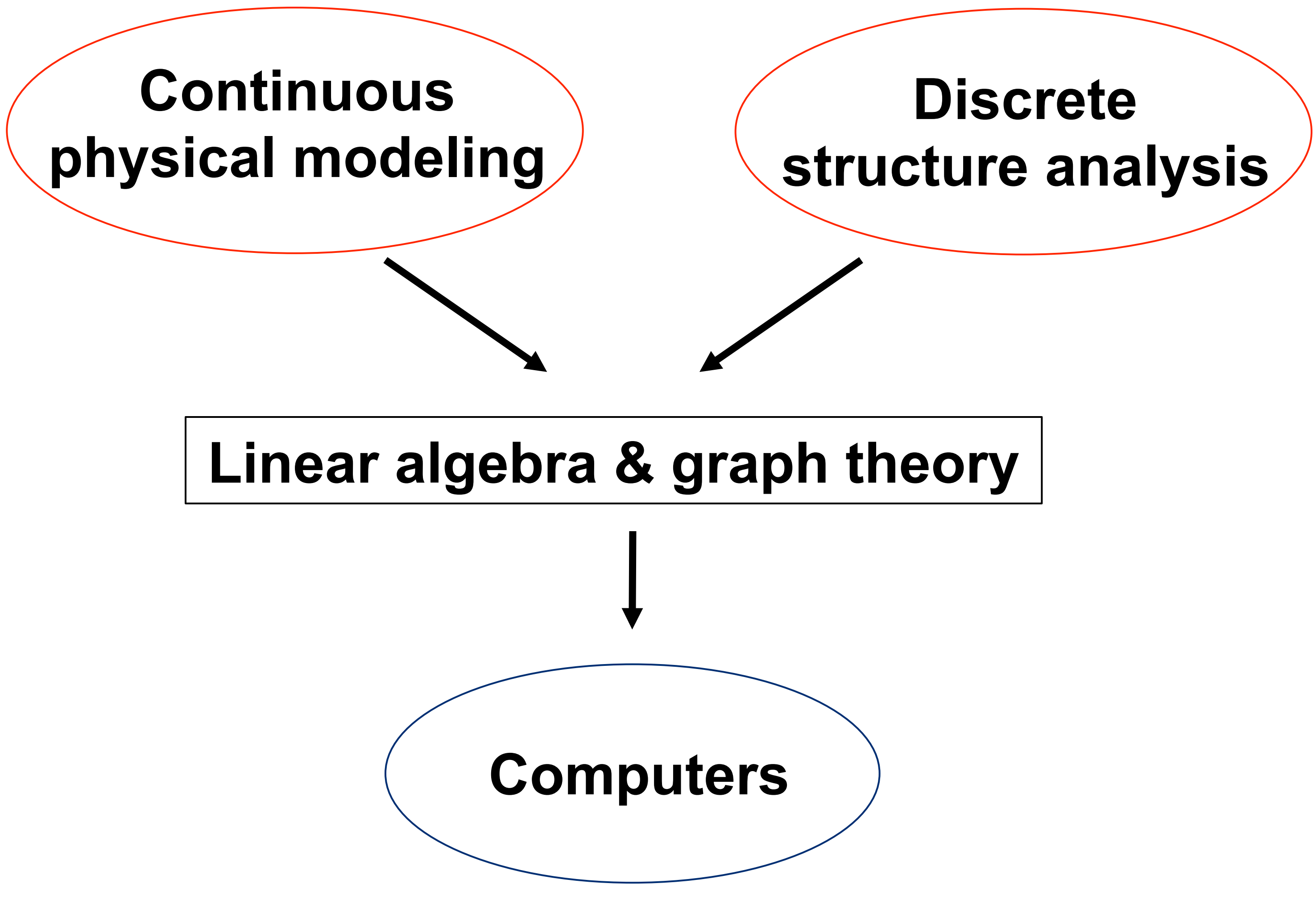
Scientific computation and data analysis



Scientific computation and data analysis

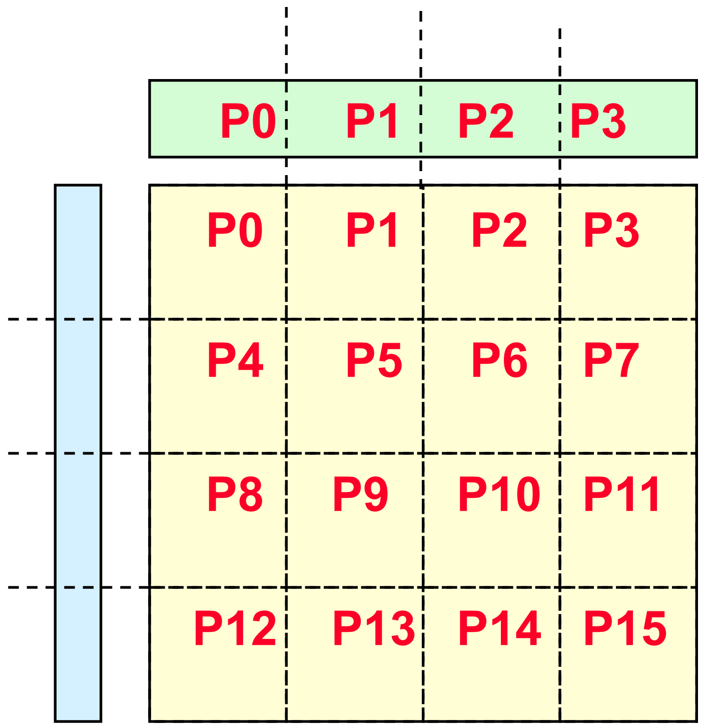


Scientific computation and data analysis



Other memory layouts for matrix-vector product

- Column layout of the matrix eliminates the broadcast
  - But adds a reduction to update the destination – same total comm
- Blocked layout uses a broadcast and reduction, both on only sqrt(p) processors – less total comm
- Blocked layout has advantages in multicore / Cilk++ too



Challenges of Irregular Meshes for PDE's

- How to generate them in the first place
  - For example, Triangle, a 2D mesh generator (Shewchuk)
  - 3D mesh generation is harder! For example, QMD (Vavasis)
- How to partition them into patches
  - For example, ParMetis, a parallel graph partitioner (Karypis)
- How to design iterative solvers
  - For example, PETSc, Aztec, Hypre   (all from national labs)
  - … Prometheus, a multigrid solver for finite elements on irregular meshes
- How to design direct solvers
  - For example, SuperLU, parallel sparse Gaussian elimination
- These are challenges to do sequentially, more so in parallel