Outline

- Incredibly brief review of simulation
- On to Differential Equations: ODEs and PDEs
- A brief look at tree algorithms (and Scan)

Simulation outline

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

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

Review of last lecture

- 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

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

Circuit Example

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

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

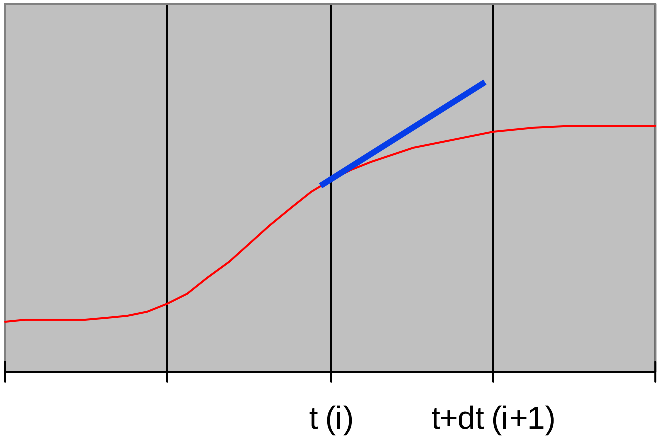
- Assume ODE is x’(t) = f(x) = A*x(t), where A is a sparse matrix
  - Compute x(i*dt) = x[i] at i=0,1,2,…
  - ODE gives x’(i*dt) = slope

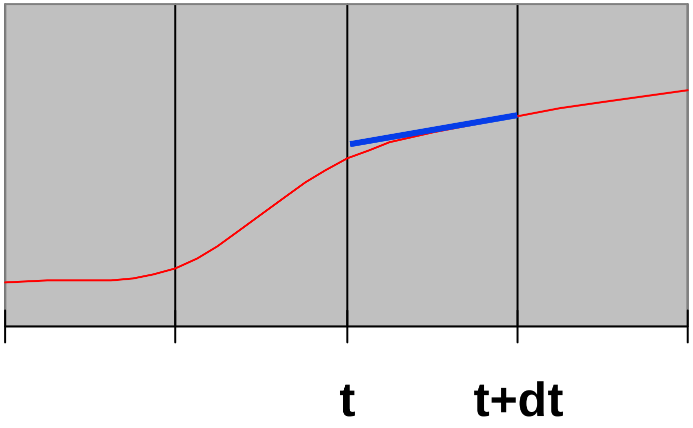
    x[i+1]=x[i] + dt*slope
- Explicit methods, e.g., (Forward) Euler’s method.
  - Approximate x’(t)=A*x(t) by (x[i+1] - x[i])/dt = A*x[i].
  - x[i+1] = x[i]+dt*A*x[i], i.e. sparse matrix-vector multiplication.
- Tradeoffs:
  - Simple algorithm: sparse matrix vector multiply.
  - Stability problems: May 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
- 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

Explicit vs. Implicit





Forward (Explicit: dt(i+1)) vs. Backward (Implicit): dt

Solving ODEs: Eigenvalue methods

- Computing modes of vibration: finding eigenvalues and eigenvectors.
  - Seek solution of $d^2\ x(t)/dt^2 = A*x(t)$ 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 $-\omega^2 *x_0 = A*x_0$, so that $-\omega^2$ is an eigenvalue and x_0 is an eigenvector of A.
  - Solution schemes reduce either to sparse-matrix multiplication, or solving sparse linear systems.

Implicit methods: Eigenproblems

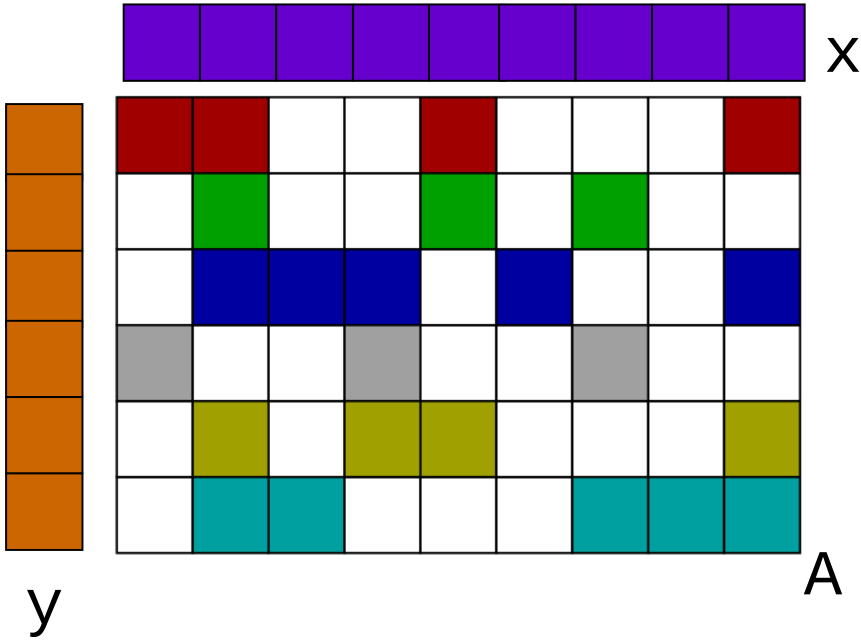
- Implicit methods for ODEs need to solve linear systems
- Direct methods (Gaussian elimination)
  - Called LU Decomposition, because we factor A = L*U
  - More complicated than sparse-matrix vector multiplication.
- Iterative solvers
  - Jacobi, Successive over-relaxation (SOR) , Conjugate Gradient (CG), Multigrid,...
  - Most have sparse-matrix-vector multiplication
- Eigenproblems
  - Also depend on sparse-matrix-vector multiplication, direct methods.

ODEs and Sparse Matrices

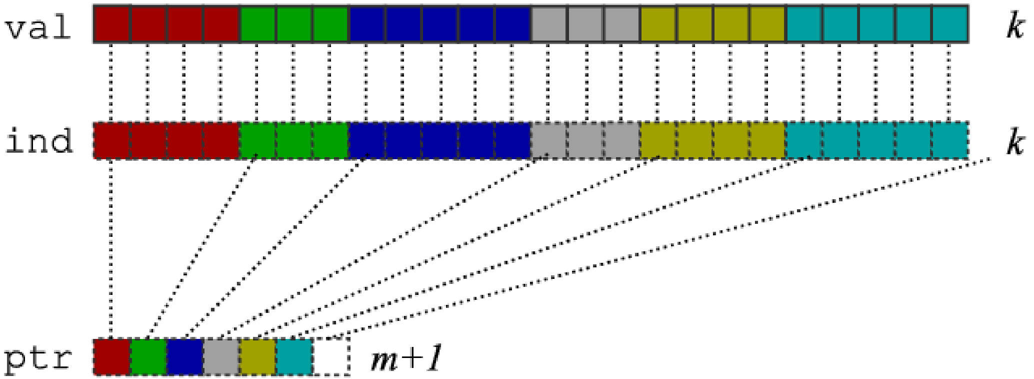
- All these problems reduce to sparse matrix problems
  - Explicit: sparse matrix-vector multiplication
  - Implicit: solve a sparse linear system
    - direct solvers (Gaussian elimination).
    - iterative solvers (use sparse matrix-vector multiplication).
  - Eigenvalue/vector algorithms may also be explicit or implicit.
- Conclusion: *Sparse Matrix-Vector Multiplication is key* to many ODE problems
  - Relatively simple algorithm to study in detail
  - Two key problems: locality and load balance

Compressed Sparse Row (CSR) Format

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



Representation of A

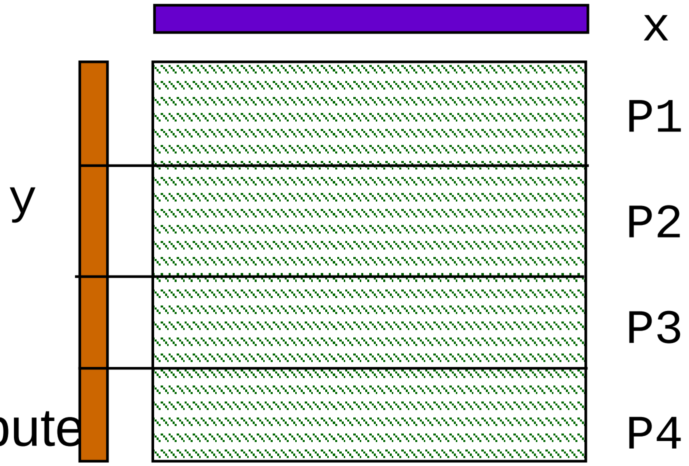


Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j) \cdot x(j)$

```
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 = A*x, where A is a sparse n x n matrix

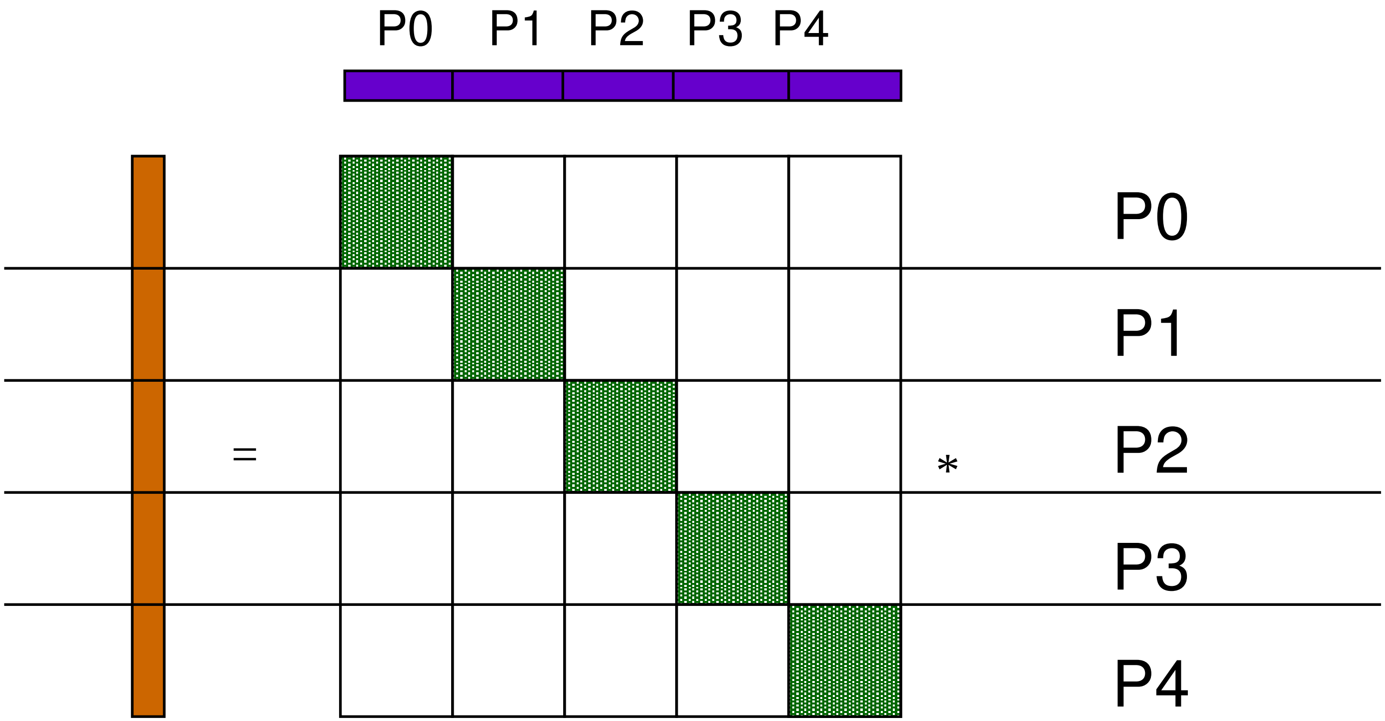


- Questions
  - which processors store
    - y[i], x[i], and A[i,j]
  - which processors compute
    - y[i] = sum (from 1 to n) A[i,j] * x[j]
      = (row i of A) * x … a sparse dot product
- Partitioning
  - Partition index set {1,…,n} = N1 ∪ N2 ∪ … ∪ Np.
  - For all i in Nk, Processor k stores y[i], 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



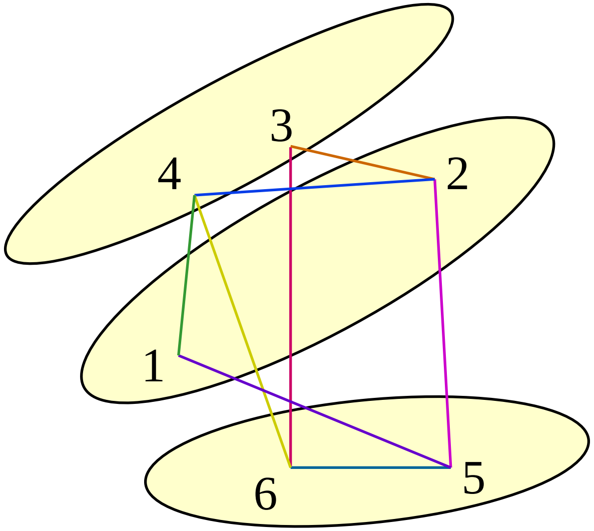
Goals of Reordering

- 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

- Relationship between matrix and graph

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

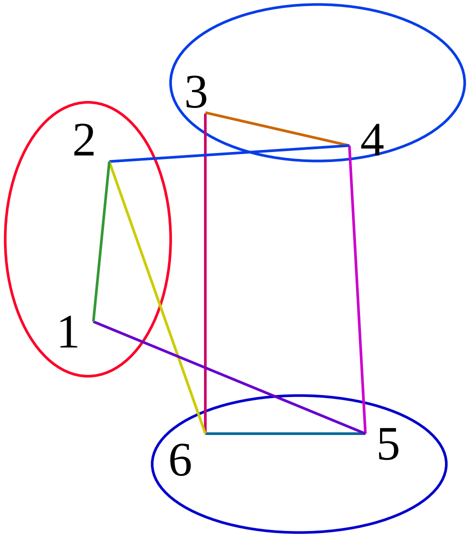


- Edges in the graph are nonzero in the matrix: here the matrix is symmetric (edges are unordered) and weights are equal (1)
- If divided over 3 processors, there are 14 nonzeros outside the diagonal blocks, which represent the 7 (bidirectional) edges

Graph Partitioning and Sparse Matrices

- Relationship between matrix and graph

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

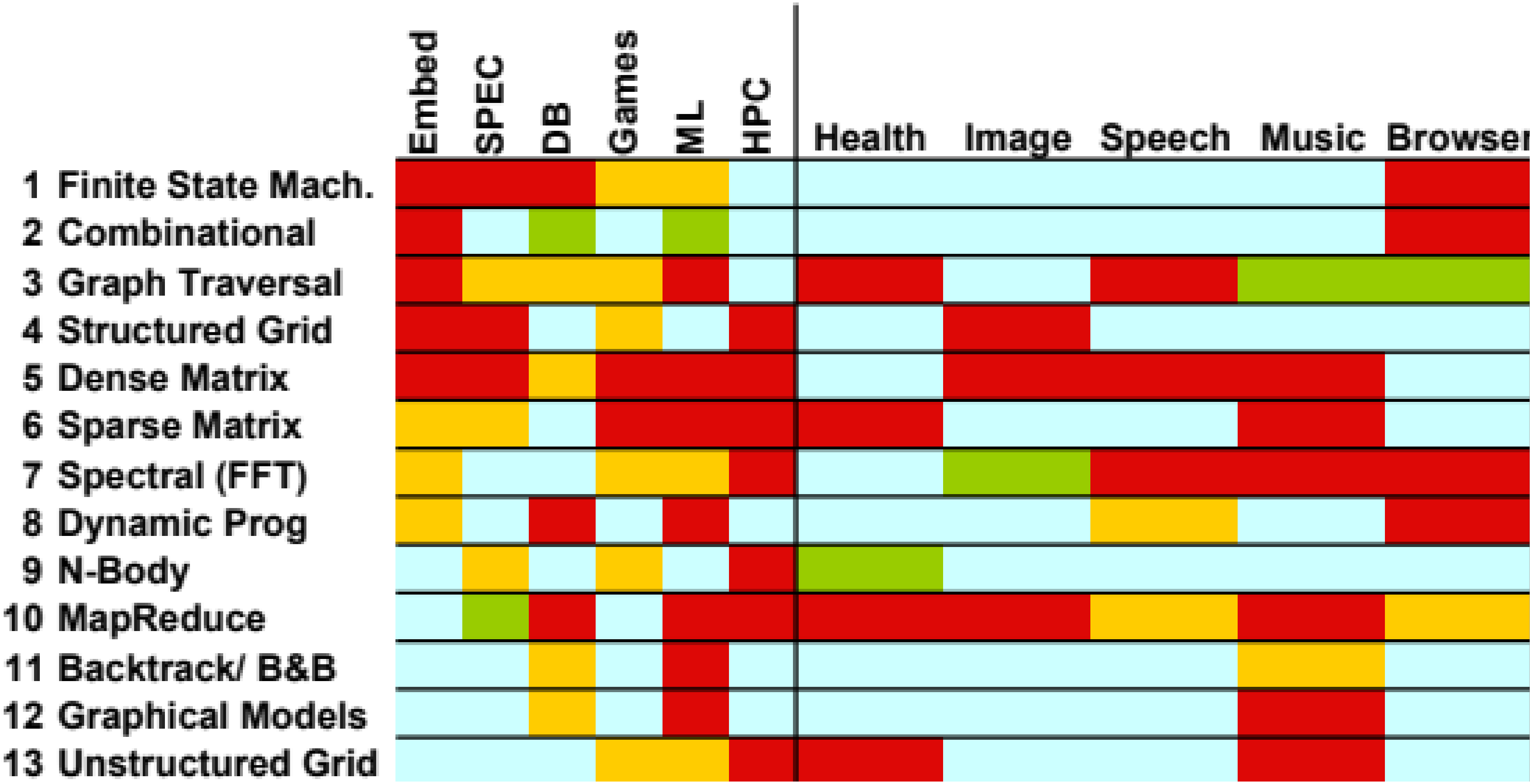


- A “good” partition of the graph has
  - equal (weighted) number of nodes in each part (load and storage balance).
  - minimum number of edges crossing between (minimize communication).
- Reorder the rows/columns by putting all nodes in one partition together.

Summary of common problems

- Load Balancing
  - Dynamically – if load changes significantly during job
  - Statically - Graph partitioning
    - Discrete systems
    - Sparse matrix vector multiplication
- 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)

Computational methods in Applications



	Embed	SPEC	DB	Games	ML	HPC	Health	Image	Speech	Music	Browser
1 Finite State Mach.	🟥	🟥	🟥	🟨	🟨	🟦	🟦	🟦	🟦	🟦	🟥
2 Combinational	🟥	🟦	🟩	🟦	🟩	🟦	🟦	🟦	🟦	🟦	🟥
3 Graph Traversal	🟥	🟨	🟨	🟨	🟥	🟦	🟥	🟦	🟥	🟩	🟩
4 Structured Grid	🟥	🟥	🟦	🟨	🟦	🟥	🟦	🟥	🟦	🟦	🟦
5 Dense Matrix	🟥	🟥	🟨	🟥	🟥	🟥	🟦	🟥	🟥	🟥	🟦
6 Sparse Matrix	🟨	🟨	🟦	🟥	🟥	🟥	🟥	🟦	🟦	🟥	🟦
7 Spectral (FFT)	🟨	🟦	🟦	🟨	🟨	🟥	🟦	🟩	🟥	🟥	🟥
8 Dynamic Prog	🟨	🟦	🟥	🟦	🟥	🟦	🟦	🟦	🟨	🟦	🟥
9 N-Body	🟦	🟨	🟦	🟨	🟦	🟥	🟩	🟦	🟦	🟦	🟦
10 MapReduce	🟦	🟩	🟥	🟦	🟥	🟥	🟥	🟥	🟨	🟥	🟨
11 Backtrack/ B&B	🟦	🟦	🟨	🟦	🟥	🟦	🟦	🟦	🟦	🟨	🟦
12 Graphical Models	🟦	🟦	🟨	🟦	🟥	🟦	🟦	🟦	🟦	🟥	🟦
13 Unstructured Grid	🟦	🟦	🟦	🟨	🟨	🟥	🟥	🟦	🟦	🟥	🟦

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)

PDEs: Local/Global Dependence

- 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)

Explicit time stepping

- Approximate PDE by ODE system (“method of lines”):
- Need a time-stepping scheme for the ODE: Simplest scheme is Euler’s Method
- Taking a time step ≡ sparse matrix vector multiplication
- This may not end well… (instability)

Implicit time stepping

- 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)

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.

Implicit vs. Explicit

- Explicit:
  - Propagates information at finite rate
  - Steps look like sparse matrix-vector (in linear case)
  - Stable step determined by fastest time scale
  - Works fine for hyperbolic PDEs
- Implicit:
  - No need to resolve fastest time scales
  - Steps can be long... but expensive
  - Linear/nonlinear solves at each step
  - Often these solves involve sparse matrix-vectors
  - Critical for parabolic PDEs

Algorithm overview

from slowest to fastest on sequential machines

- Dense LU: Gaussian elimination; works on any N-by-N matrix.
- Band LU: Exploits the fact that tridiagonal matrix 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).

Summary of Approaches to Solving PDEs

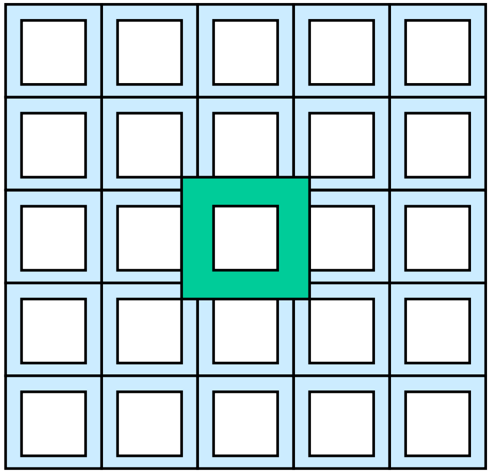
- As with ODEs, either explicit or implicit approaches are possible
  - Explicit, sparse matrix-vector multiplication
  - Implicit, sparse matrix solve at each step
    - Direct solvers are hard
    - Iterative solves turn into sparse matrix-vector multiplication: Graph partitioning
- Graph and sparse 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

Parallelism in Regular meshes

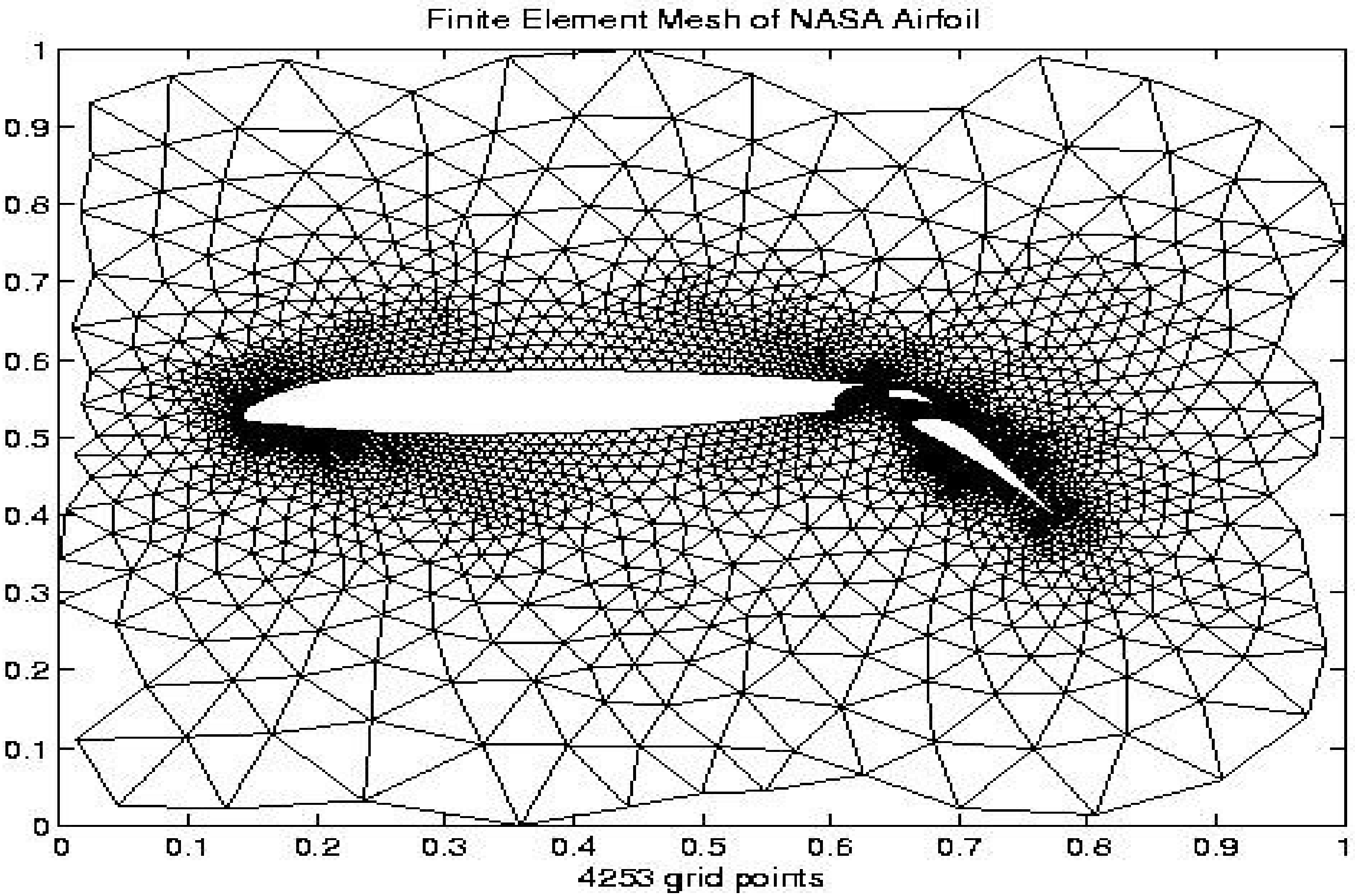
- Computing a Stencil on a regular mesh
  - need to communicate mesh points near boundary to neighboring processors.
    - Often done with ghost regions
- Surface-to-volume ratio keeps communication down, but
  - Still may be problematic in practice



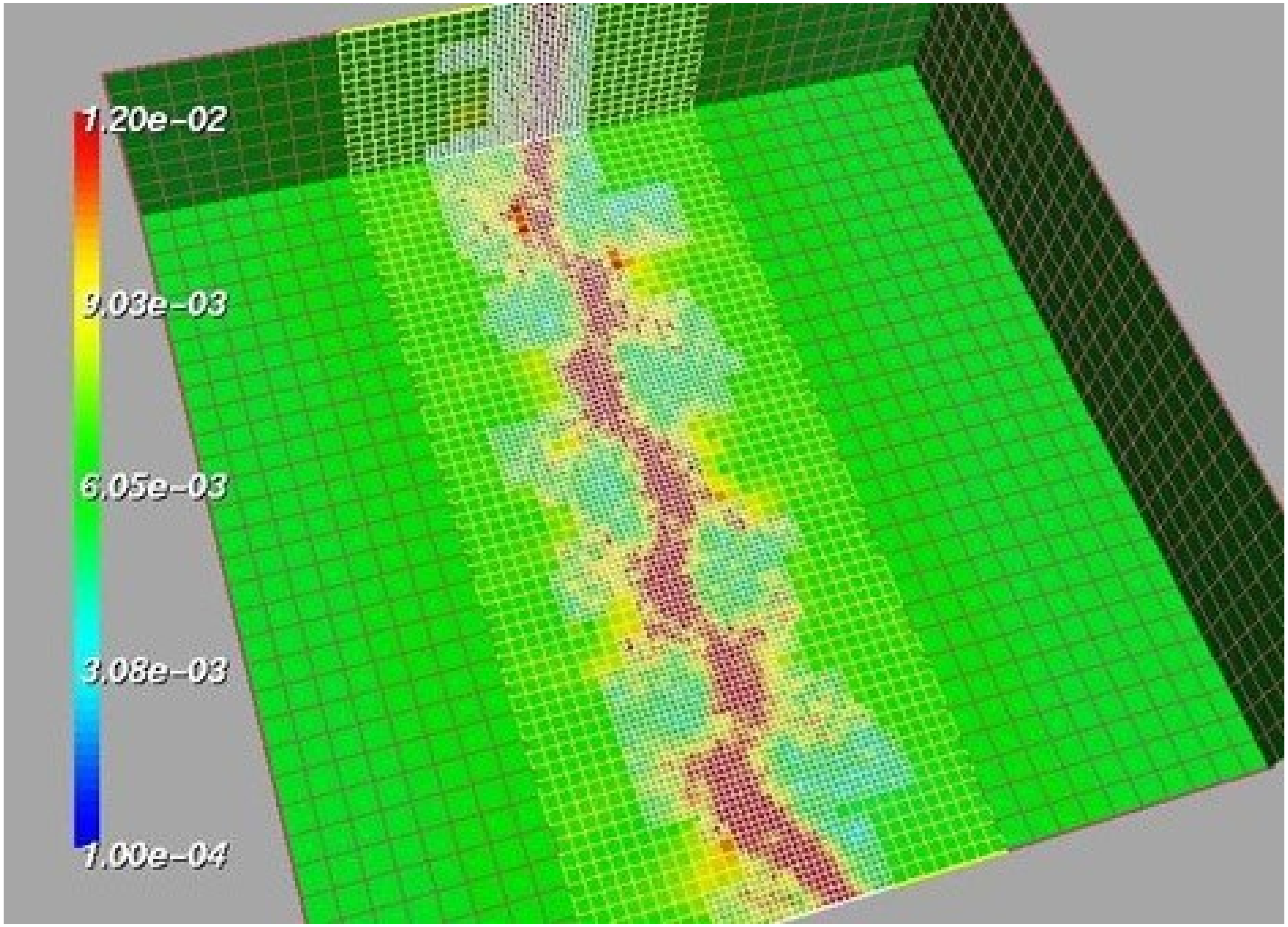
Implemented using “ghost” regions.

Adds memory overhead

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



Adaptive mesh



Challenges of Irregular Meshes

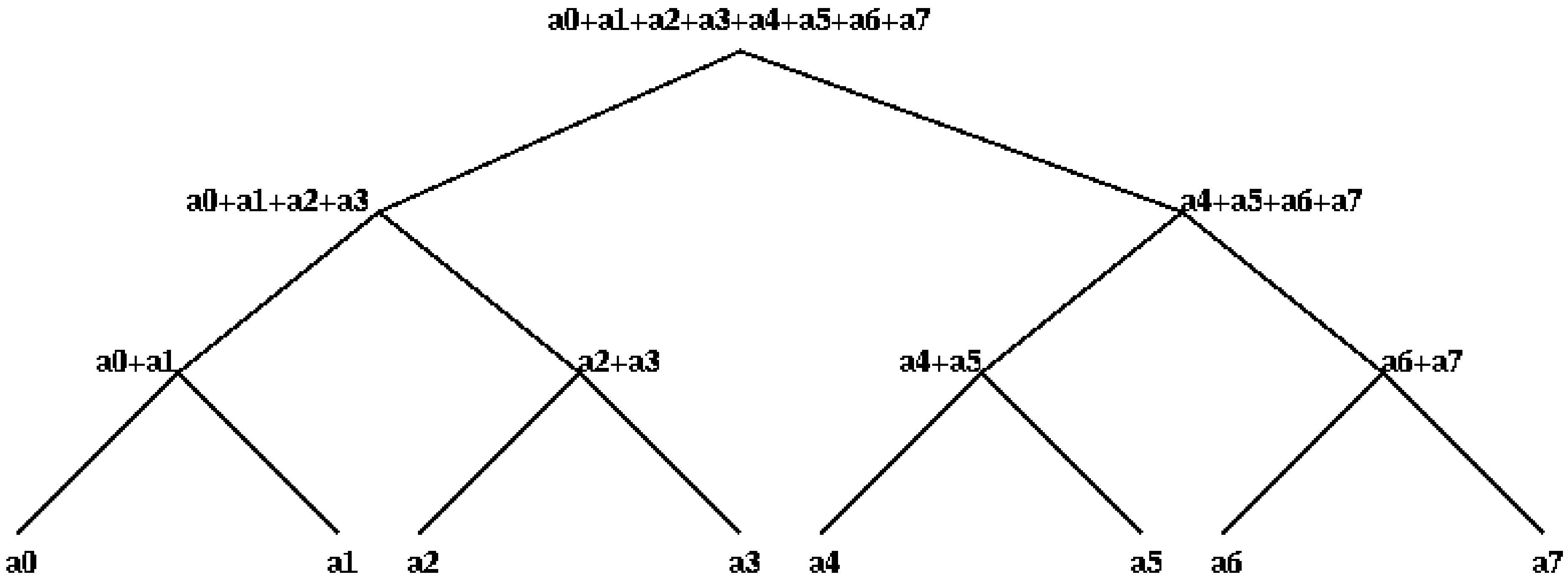
- How to generate them in the first place
  - Start from geometric description of object
    - Triangle, a 2D mesh partitioner by Jonathan Shewchuk
    - 3D harder!
- How to partition them
  - ParMetis, a parallel graph partitioner
- How to design iterative solvers
  - PETSc, a Portable Extensible Toolkit for Scientific Computing
  - Prometheus, a multigrid solver for finite element problems on irregular meshes
- How to design direct solvers
  - SuperLU, parallel sparse Gaussian elimination

The “Seven Dwarfs”:

High-end simulation in the physical sciences

1) Structured grids
2) Unstructured grids
3) Spectral methods (Fast Fourier Transform)
4) Dense Linear Algebra
5) Sparse Linear Algebra: Both explicit and implicit
6) Particle Methods
7) Monte Carlo/Embarrassing Parallelism/Map Reduce (easy!)

Tree structured computation

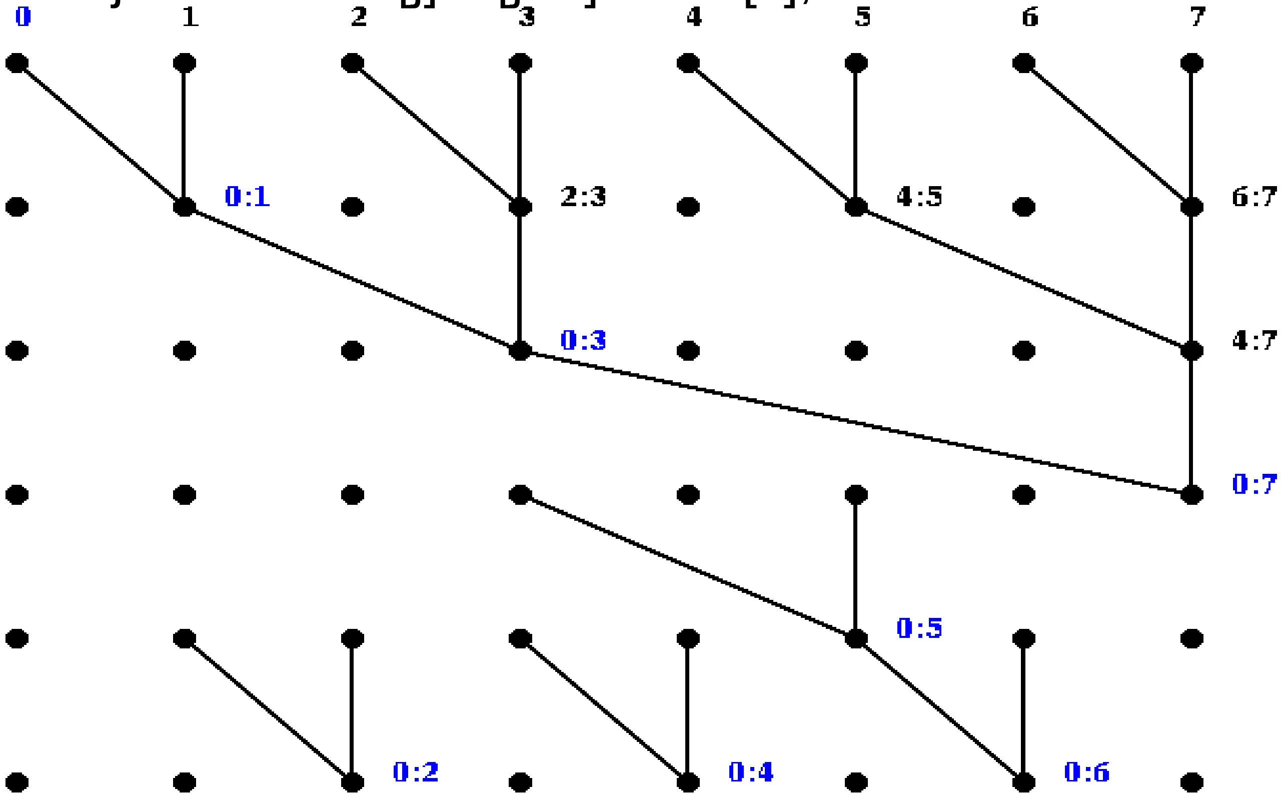


Parallel Prefix, or Scan

- If “+” is an associative operator, and x[0],…,x[p-1] are input data then parallel prefix operation computes

  y[j] = x[0] + x[1] + … + x[j]   for j=0,1,…,p-1

- Notation: j:k means x[j]+x[j+1]+…+x[k], blue is final value



Mapping Parallel Prefix onto a Tree: Details

- Up-the-tree phase (from leaves to root)

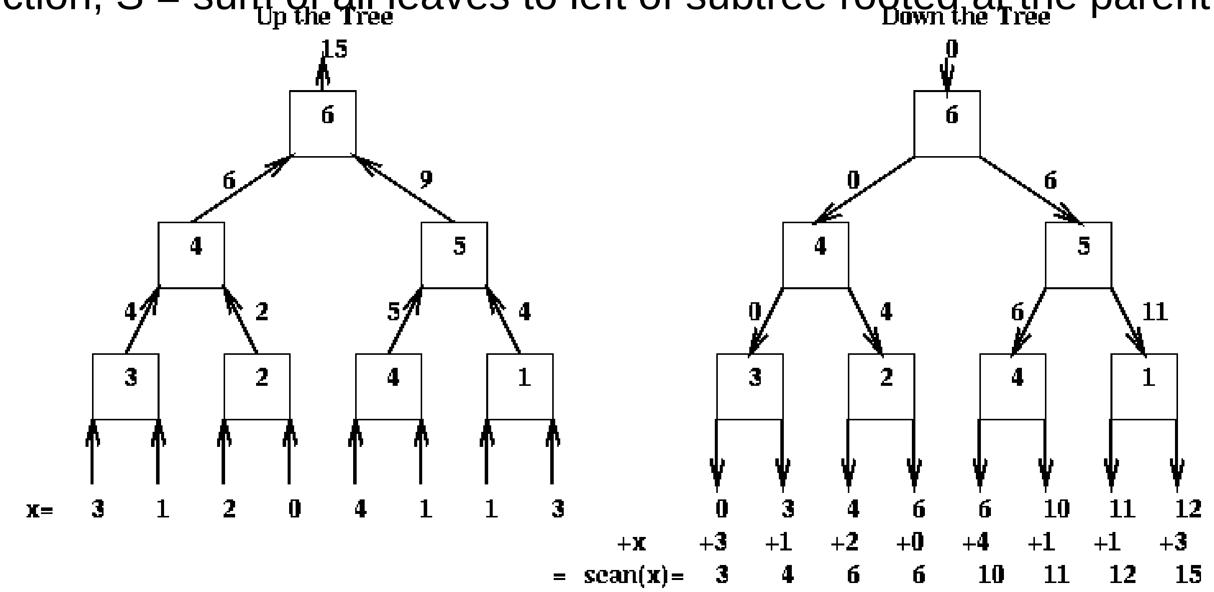
  1) **Get values L and R from left and right children**
  2) **Save L in a local register Lsave**
  3) **Pass sum L+R to parent**

- By induction, Lsave = sum of all leaves in left subtree

- Down the tree phase (from root to leaves)

  1) Get value S from parent (the root gets 0)
  2) Send S to the left child
  3) Send S + Lsave to the right child

- By induction, S = sum of all leaves to left of subtree rooted at the parent



Adding two n-bit ints in O(log n) time

- Let a = a[n-1]a[n-2]…a[0] and b = b[n-1]b[n-2]…b[0] be two n-bit binary numbers

- We want their sum s = a+b = s[n]s[n-1]…s[0]

```
c[-1] = 0          … rightmost carry bit
for i = 0 to n-1
    c[i] = ( (a[i] xor b[i])  and  c[i-1] )  or  ( a[i]  and  b[i] )   ... next carry bit
    s[i] = ( a[i] xor b[i] ) xor c[i-1]
```

- Challenge: compute all c[i] in O(log n) time via parallel prefix

  for all (0 <= i <= n-1) p[i] = a[i] xor b[i]   … propagate bit

  for all (0 <= i <= n-1) g[i] = a[i] and b[i]   … generate bit

$$\begin{pmatrix} c[i] \\ 1 \end{pmatrix} = \begin{pmatrix} (\ p[i] \text{ and } c[i-1]\ ) \text{ or } g[i] \\ 1 \end{pmatrix} = \begin{pmatrix} p[i] & g[i] \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} c[i-1] \\ 1 \end{pmatrix} = C[i] * \begin{pmatrix} c[i-1] \\ 1 \end{pmatrix}$$

… 2-by-2 Boolean matrix multiplication (associative)

$$= C[i] * C[i-1] * \ldots C[0] * \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

… evaluate each P[i] = C[i] * C[i-1] * … * C[0] by parallel prefix

Used in all computers to implement addition - Carry look-ahead

Browser page layout via Prefix (Scan)

- Applying layout rules to html description of a webpage is a bottleneck, scan can help
- Simplest example
  - Given widths [x_1, x_2, … , x_n] of items to display on page, where should each item go?
  - Item j starts at $x_1 + x_2 + … + x_{j-1}$
- Real examples have complicated constraints
  - Defined by general trees, since in html each object to display can be composed of other objects
  - To get location of each object, need to do preorder traversal of tree, “adding up” constraints of previous objects
  - Scan can do preorder traversal of any tree in parallel
    - Not just binary trees

Summary of tree algorithms

- Lots of problems can be done quickly - in theory - using trees
- Some algorithms are widely used
  - broadcasts, reductions, parallel prefix
  - carry look ahead addition
- Some are of theoretical interest only
  - Csanky’s method for matrix inversion
  - Solving tridiagonal linear systems (without pivoting)
  - Both numerically unstable
- Embedded in various systems
  - MPI, NESL (CMU), other languages
  - CM-5 hardware control network