Methods of Computational Astrophysics: Galaxy Simulation

Nathaniel R. Stickley

Department of Physics & Astronomy

UC RIVERSIDE UNIVERSITY OF CALIFORNIA

Outline

The Problem

- We try to understand *dynamics* by looking at *static* images.
- We can view different objects at different epochs, but we cannot follow the evolution of individual objects.
- It is usually impossible to perform experiments in a laboratory to better understand cosmological objects.

The Problem

- We try to understand *dynamics* by looking at *static* images.
- We can view different objects at different epochs, but we cannot follow the evolution of individual objects.
- It is usually impossible to perform experiments in a laboratory to better understand cosmological objects.

Idea:

Use known physics to build a computational model of the system of interest. Watch the simulated objects evolve in the model.

The Ideal Solution

If we had infinite computing power available...

- Create fully-realistic simulation.
- Calculate all known physics.
- Precision is limited only by uncertainty principles.
- Simulate individual photons, gas, dust particles.
  - Stellar evolution is computed *automatically* and consistently, just as in nature.

The First Approximation

- Approximate stars as spheres or point-masses.
- Approximate electrodynamics using classical electrodynamics.
- Approximate gases and plasmas as continua.
- From the distribution of matter and EM field, construct the energy-momentum tensor $T_{\mu\nu}$
- Solve the Einstein field equation...

The First Approximation

The Einstein field equation

$$G_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$$

where

$$G_{\mu\nu} = R_{\mu\nu} - \frac{1}{2} g_{\mu\nu} R$$

$$\Rightarrow R_{\mu\nu} - \frac{1}{2} g_{\mu\nu} R = \frac{8\pi G}{c^4} T_{\mu\nu}$$

Ricci tensor $R_{\mu\nu}$ and scalar curvature R are constructed from derivatives of the metric $g_{\mu\nu}$.

Goal: Find the metric $g_{\mu\nu}$.

The First Approximation

- Once metric is obtained, evolve matter and fields forward.
- Repeat process.

Problem: System of 10 non-linear, coupled PDE's!

$$T_{\mu\nu} = \begin{pmatrix} T_{00} & T_{01} & T_{02} & T_{03} \\ T_{10} & T_{11} & T_{12} & T_{13} \\ T_{20} & T_{21} & T_{22} & T_{23} \\ T_{30} & T_{31} & T_{32} & T_{33} \end{pmatrix}, \quad T_{\mu\nu} = T_{\nu\mu}$$

Possible to solve numerically, but very difficult task in general.

The First Approximation

- Once metric is obtained, evolve matter and fields forward.
- Repeat process.

Problem: System of 10 non-linear, coupled PDE's!

$$T_{\mu\nu} = \begin{pmatrix} T_{00} & T_{01} & T_{02} & T_{03} \\ T_{10} & T_{11} & T_{12} & T_{13} \\ T_{20} & T_{21} & T_{22} & T_{23} \\ T_{30} & T_{31} & T_{32} & T_{33} \end{pmatrix}, \quad T_{\mu\nu} = T_{\nu\mu}$$

Possible to solve numerically, but very difficult task in general.

Code: OpenGR

`http://wwwrel.ph.utexas.edu/openGR/`

The Second Approximation

Rather than solving the full GR field equations...

1. Linearize the field equations OR...
2. Generalize Newtonian gravitation in analogy with classical electrodynamics

Both approaches yield similar results:

Contain signal-retardation effects, velocity-dependent forces (gravimagnetic field), and radiation.

The Second Approximation

Principal application: Celestial mechanics in the Solar System

- Calculate trajectories of planets, comets, asteroids, moons
- Calculate spacecraft trajectories with high precision (GPS, for instance)
- Create astronomical almanacs / ephemerides

Secondary: Detailed models of binary star systems, 3-star interactions, other small systems.

The Third Approximation

Remove explicit velocity-dependence from equations of motion:

- Static limit of linearized GR.
- Identical to Newtonian gravitation with finite signal speed.
- Does not violate causality.
- Less computationally expensive than linearized GR.
- Requires position data to be stored for $t = D/c$ where D is size of simulation.
- No *known** codes use this method.

Direct N-Body

The 4th level of approximation: Use Newtonian gravitation

- Assumes gravitation propagates with infinite speed.
- Forces depend only on position.
- Requires only the *current* position data to be stored in memory.
- Less computationally expensive than previously-described methods, but still quite expensive...

Direct N-Body

A note on scaling: Direct N-Body scales as N^2

- For a simulation containing N bodies, there are $N\left(N-1\right)$ force pairs.
- Eliminating double-counting leaves $\frac{1}{2}N\left(N-1\right)$ force calculations of the form

$$\mathbf{F}_{ij} = \frac{-Gm_i m_j \mathbf{r}_{ij}}{r_{ij}^3}$$

In units for which $G = 1$, this is *at least* 3 multiplications and 1 exponentiation for each force calculation. (In practice, more because of the 3-component vector)

Direct N-Body

A note on scaling (cont'd):

Example 1

A simulation using $100,000$ particles requires $\gtrsim 1.5$GFlop per time-step.

GFlop = Billion Floating point operations

Direct N-Body

A note on scaling (cont'd):

Example 1

A simulation using $100,000$ particles requires $\gtrsim 1.5$GFlop per time-step.

GFlop = Billion Floating point operations

Example 2

A simulation using 10^6 particles requires $\gtrsim 1$ TFlop per time-step.

Direct N-Body

A note on scaling (cont'd):

Example 1

A simulation using $100,000$ particles requires $\gtrsim 1.5$GFlop per time-step.

GFlop = Billion Floating point operations

Example 2

A simulation using 10^6 particles requires $\gtrsim 1$ TFlop per time-step.

For a realistic galaxy, $N \sim 10^{11} - 10^{12}$, requiring $\sim 10^{11}$ TFlop per time step for the force calculation.

Direct N-Body

For comparison: The current fastest computer, K Computer:

- 705,024 processing cores (+ Nvidia Tesla GPUs).
- 8.16 petaFlop/s (8.16×10^3 TFlop/s).
- $\sim 4 - 5$ months for one iteration.

Visit `http://www.top500.org` for current list of the top 500 supercomputers in the world.

Comments:

- Very computationally expensive.
- Not practical for galaxy simulations.
- Only used in *very* high resolution simulations of small systems such as open clusters and globular clusters

Gravitational Softening

The Motivation:

- Most of the computational effort in N-body simulations is spent on binary stars and other close encounters between star particles.
- Close encounters drive two-body relaxation.

Gravitational Softening

The Motivation:

- Most of the computational effort in N-body simulations is spent on binary stars and other close encounters between star particles.
- Close encounters drive two-body relaxation.

Solution: Prevent the formation of binary star systems and close particle-particle encounters.

- Replace point-particles with extended "fuzzy" particles.
- Particles now represent small *groups* of stars rather than single stars.
- For short distances, replace Newtonian force with another expression corresponding to extended objects.

Gravitational Softening

More Details:

Point particles are "smeared out" into sphere's of radius ϵ. The parameter ϵ is known as the softening length.

Examples of Softening Methods:

Plummer Softening

Replace $\frac{Gm_i m_j \mathbf{r}_{ij}}{r_{ij}^3}$ with $\frac{Gm_i m_j \mathbf{r}_{ij}}{\left(r_{ij}^2+\epsilon^2\right)^{3/2}}$

Gravitational Softening

Plummer Softening Spline

$$\mathbf{F}_{ij} = \begin{cases} -\dfrac{Gm_i m_j \mathbf{r}_{ij}}{r_{ij}^3} & \text{for } r > \epsilon \\ -\dfrac{Gm_i m_j \mathbf{r}_{ij}}{\left(r_{ij}^2+\epsilon^2\right)^{3/2}} & \text{for } r \leq \epsilon \end{cases}$$

- Note: this particular spline is discontinuous, but it works.
- Other methods are also utilized, often incorporating Gaussians and splines.

Gravitational Softening

Comments:

- Detail about binaries is lost in the process of softening.
- Softening length provides a resolution limit.
- Allows for much higher performance than point-particle direct N-body.
- Scales as N^2, but now N represents groups of stars rather than individual stars.
- Practical for medium-to-low resolution simulations of clusters and very high resolution simulations of small galaxies.
- Too computationally expensive to be practical for simulations of large galaxies and galaxy mergers.

Barnes-Hut Tree Method

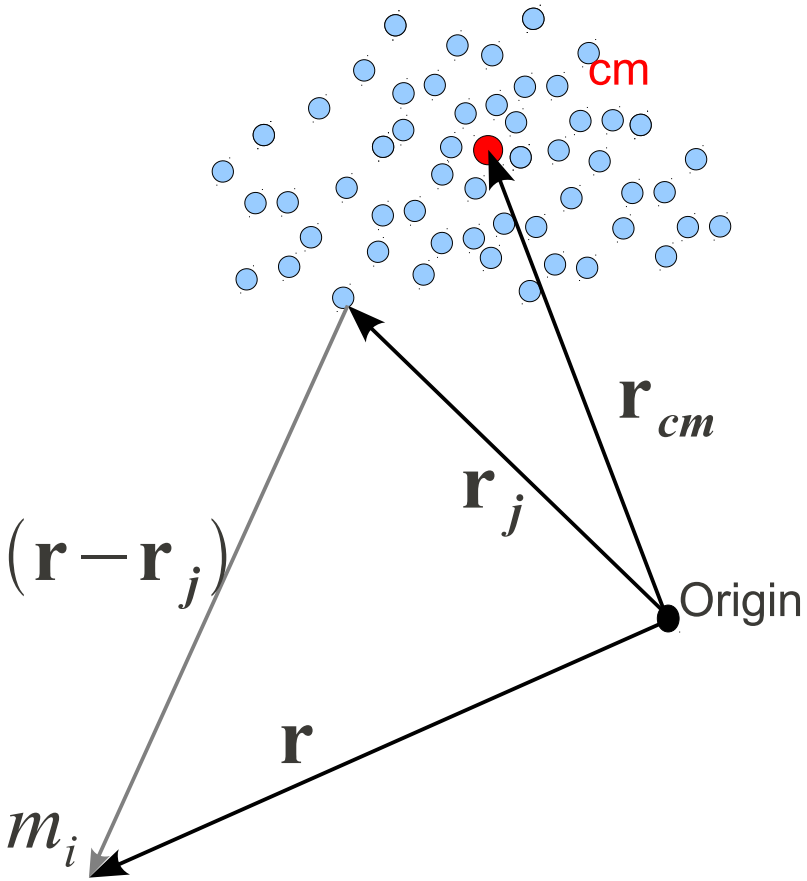
The next approximation: Use the direct N-body method only locally. Use multipole expansion for larger distances.

Steps:

1. Divide space into cubical regions.
2. Continue subdividing space until each cube only contains one (or other *small* number) of particles.
3. Calculate the multipole expansion of the gravitational field for each cube and sub-cube recursively.
4. Use direct N-body method for force due to nearest-neighbor particles.
5. Use multipole expansions to calculate force due to distant groups of particles.

Barnes-Hut Tree Method

Multipole expansion geometry:



Barnes-Hut Tree Method

Monopole moment:

$$M = \sum_j m_j$$

Quadrupole moment tensor:

$$Q_{k\ell} = \sum_j m_j \left[3 \left(\mathbf{r}_j - \mathbf{r}_{cm} \right)_k \left(\mathbf{r}_j - \mathbf{r}_{cm} \right)_\ell - \delta_{k\ell} \left(\mathbf{r}_j - \mathbf{r}_{cm} \right)^2 \right]$$

Defining $\mathbf{x} = \mathbf{r} - \mathbf{r}_{cm}$, the potential at point $\mathbf{r}$ is

$$\Phi \left(\mathbf{r} \right) = -G \left[\frac{M}{|\mathbf{x}|} + \frac{1}{2} \frac{Q_{k\ell} x^k x^\ell}{|\mathbf{x}|^5} \right]$$

In some codes, only the monopole term is used.

Barnes-Hut Tree Method

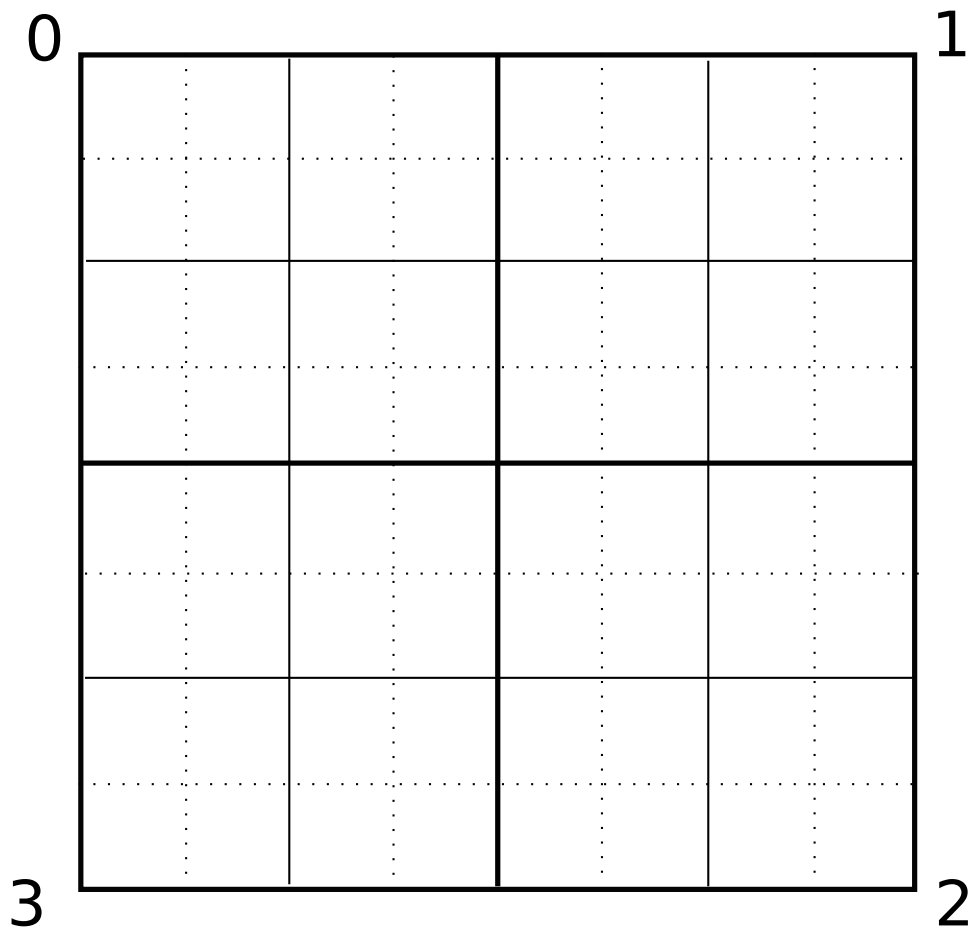
Sub-diving space

Space is subdivided using a data structure called an Oct-tree, or simply **Octree.** The octree efficiently lists:

- which particles are located in each cube,
- which sub-cubes are located within other sub-cubes,
- which cubes are neighbors,
- multipole moments for each sub-cube.
- coordinates of the center of mass of each sub-cube.

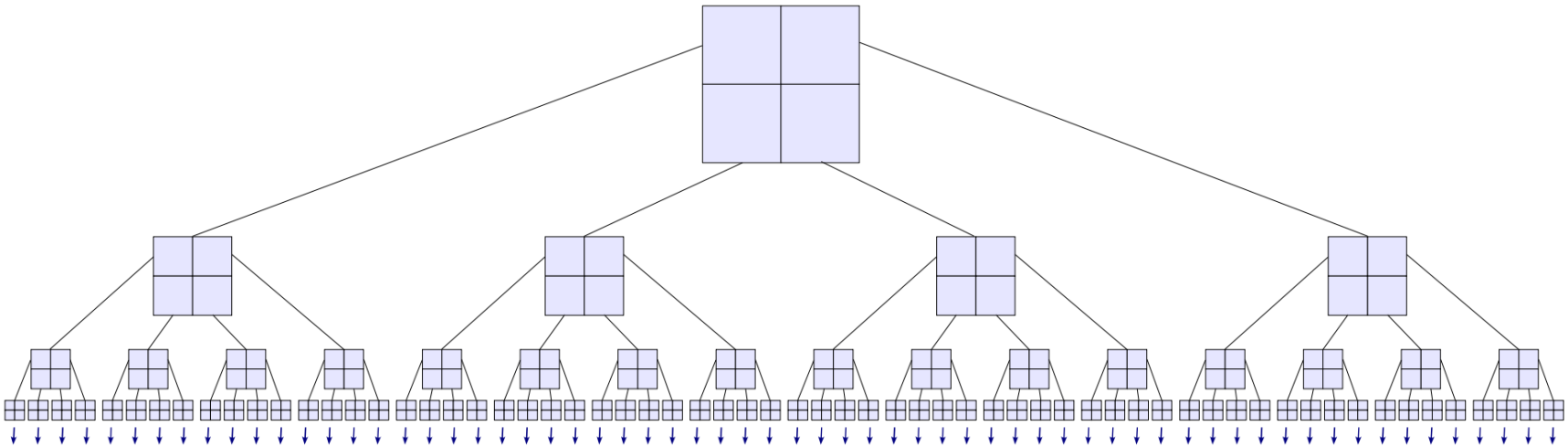
Barnes-Hut Tree Method

Since 2-D is easier to represent graphically, I'll explain quad-trees:



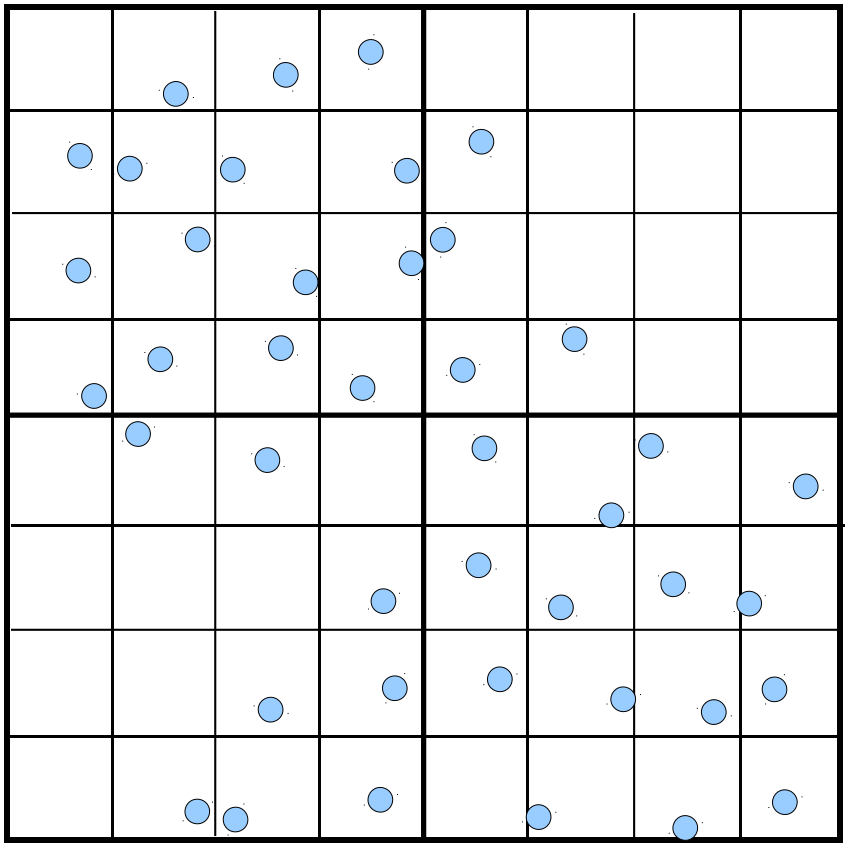
Barnes-Hut Tree Method

The corresponding quad-tree



Barnes-Hut Tree Method

A populated grid:



Barnes-Hut Tree Method

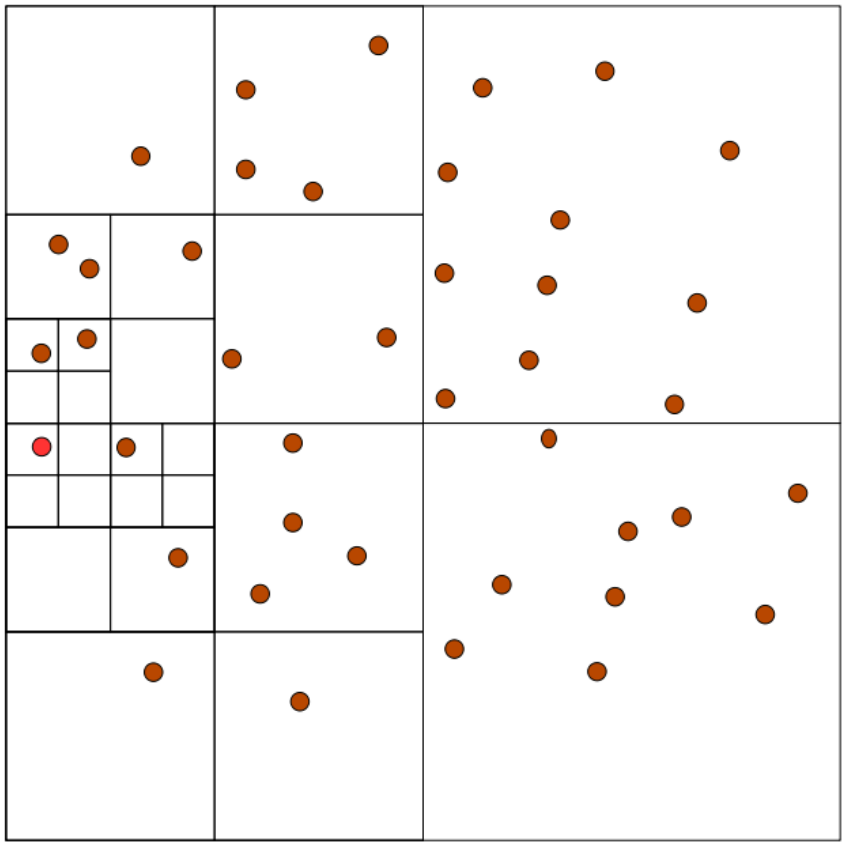
We need to calculate forces using direct N-body method close to each point and multipole approximations for larger distances. Thus we need to:

- Decide how to separate "local" from "distant" particles.
- Decide how to ensure accuracy of multipole expansions while minimizing the computational effort.

Intuitively, small volumes should be used near the particle of interest and the size of the volumes can be larger as the separation distance increases...

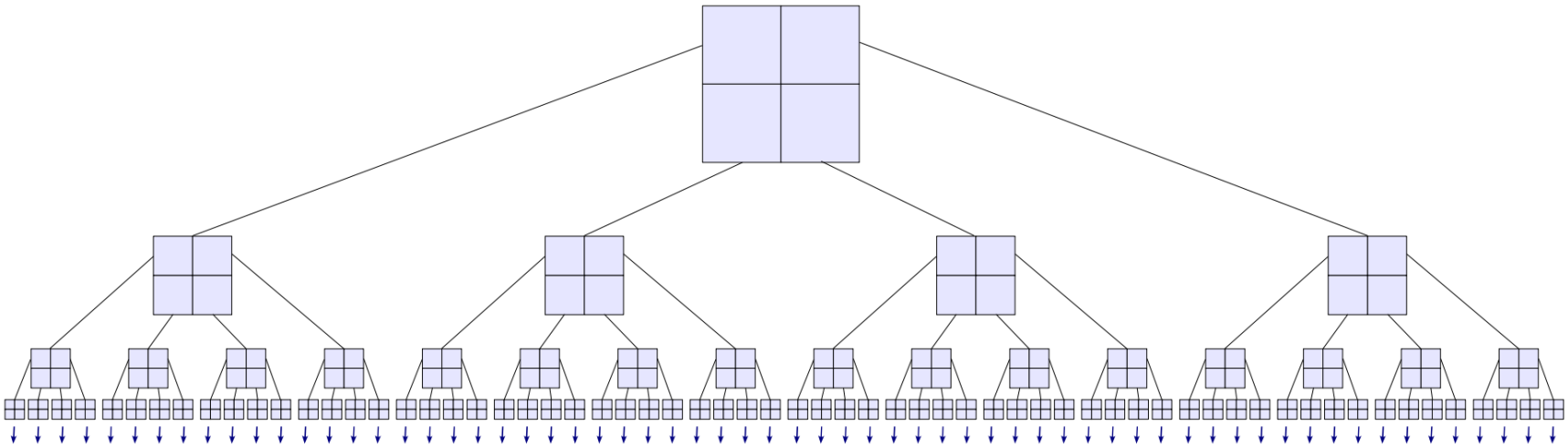
Barnes-Hut Tree Method

For one particle:



Barnes-Hut Tree Method

- The process of calculating the forces and deciding how to calculate forces is called “walking the tree”.
- The “opening condition” tells us whether we need to descend down a branch



Barnes-Hut Tree Method

The most common opening condition:

- The **opening angle** criterion is based on the approximate angle subtended by the node.

$$\theta \equiv \frac{d}{r} \leq \theta_c$$

- d: dimension of the node (length of cube side).
- r : distance to the cm of the node.
- θ_c: critical opening angle.
- Accuracy of the code is determined by θ_c. (smaller $\theta_c \Rightarrow$ less error)

Barnes-Hut Tree Method

Comments:

- Accuracy depends on opening condition.
- For very strict opening conditions, tree method is identical to direct N-body
- Scales as $N \log N$ (assuming a reasonable opening condition)
  - For 10^6 particles, tree method requires $\sim 10^8$Flop
  - The cost: much larger memory requirement
  - Greater code complexity
- This is the method typically used for galaxy-scale and cluster-scale simulations.

Particle-Mesh Method

An alternative approach: Recall that, for Newtonian gravity,

$$\nabla^2 \Phi = 4\pi\rho$$

- Rather than calculating force at the location of each particle, use this to calculate the acceleration field at each point on a Cartesian mesh.
- Interpolate acceleration values from the mesh to the position of each particle.
- If the number of mesh points N_m is much smaller than the number of Particles N, this will lead to a significant increase in efficiency.

Particle-Mesh Method

Basic procedure:

1. Convert discrete particle distribution to density function, ρ, using interpolation.
2. Solve
$$\nabla^2 \Phi = 4\pi\rho$$
to obtain Φ at each mesh point.
3. Use finite difference method to compute acceleration at each mesh point.
4. Transfer the acceleration field from the mesh points back to particles using interpolation.

Particle-Mesh Method

Two common interpolation methods: Nearest-neighbor (NN) and cloud-in-cell (CIC)

For NN mass - density interpolation (in 2-D), the density is transferred to the mesh as follows:

- Each particle of mass m_j has a density contribution

$$\rho_j = \frac{m_j}{h^2}$$

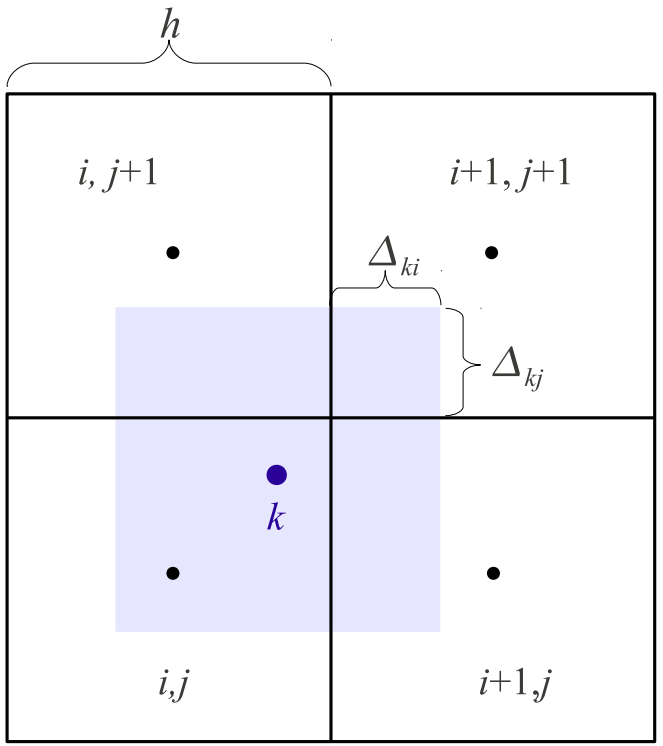
  where h is the mesh spacing.
- This contribution is added to the mesh point closest to the particle.

Particle-Mesh Method

The CIC interpolation method is slightly more complicated, but yields better results.

- The coordinates of the center of the cell i, j are denoted (x_i, y_j)
- The coordinates of a particle, k (the blue dot) are (x_k, y_k)
- Define

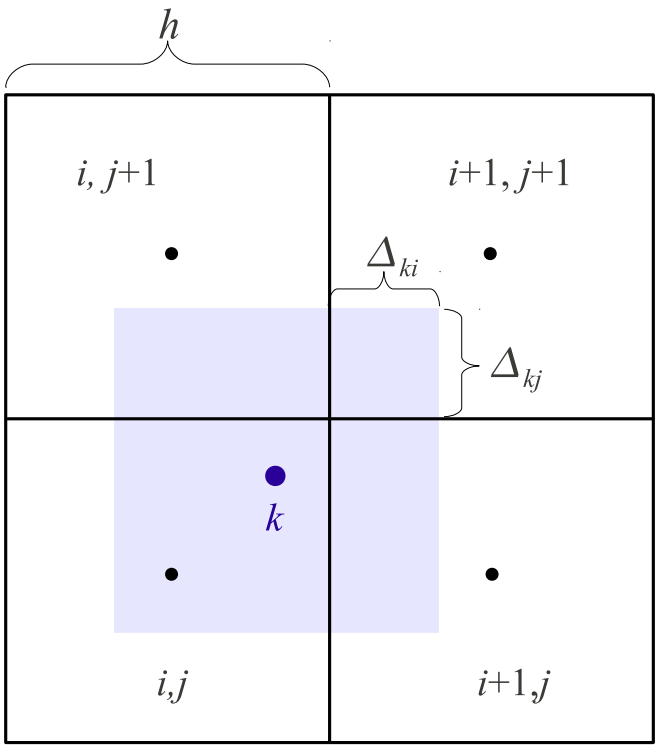
$$\Delta_{ki} \equiv x_k - x_i \quad , \quad \Delta_{kj} \equiv y_k - y_j$$



Particle-Mesh Method

- For particle of mass m_k located at (x_k, y_k), the density is given by

$$\rho_{ij} = \frac{m_k}{h^4}\left(h - \Delta_{ki}\right)\left(h - \Delta_{kj}\right)$$
$$\rho_{i+1,j} = \frac{m_k}{h^4}\Delta_i\left(h - \Delta_{kj}\right)$$
$$\rho_{i,j+1} = \frac{m_k}{h^4}\left(h - \Delta_{ki}\right)\Delta_{kj}$$
$$\rho_{i+1,j+1} = \frac{m_k}{h^4}\Delta_{ki}\Delta_{kj}$$



Particle-Mesh Method

Compute Φ at the cell centers

There are many methods available. Some include:

- Jacobi method
- Gauss-Seidel
- Successive over-relaxation
- Fourier convolution
  - Most commonly used because of FFT performance

Particle-Mesh Method

Compute $\mathbf{g}$ at cell centers

- Calculate $\mathbf{g} = -\nabla\Phi$, or more explicitly, $-\left(\frac{\partial\Phi}{\partial x_i}, \frac{\partial\Phi}{\partial y_j}, \frac{\partial\Phi}{\partial z_k}\right)$ at cell centers using a finite difference method.
- For instance, the centered difference:

$$\frac{\partial\Phi}{\partial x_i} = \frac{\Phi\left(x_{i+1}\right) - \Phi\left(x_{i-1}\right)}{2h} + \mathcal{O}(h^3)$$

Particle-Mesh Method

Interpolate g from mesh to particles

For NN interpolation,

$$\mathbf{F}_k = \mathbf{F}_{ij}$$

For the CIC interpolation,

$$\mathbf{F}_k = \kappa_1 \mathbf{F}_{ij} + \kappa_2 \mathbf{F}_{i+1,j} + \kappa_3 \mathbf{F}_{i,j+1} + \kappa_4 \mathbf{F}_{i+1,j+1}$$

$$\begin{aligned}
\kappa_1 &= (h - \Delta_{ki})(h - \Delta_{kj}) \\
\kappa_2 &= \Delta_{ki}(h - \Delta_{kj}) \\
\kappa_3 &= (h - \Delta_{ki})(h - \Delta_{kj}) \\
\kappa_4 &= \Delta_{ki}\Delta_{kj}
\end{aligned}$$

Particle-Mesh Method

Comments:

- Scales as $N_m \log N_m$
- Performance gain is large if $N_m \ll N$ (i.e. many particles per mesh point)
- Resolution is limited by spacing h.
- This is the method used for cosmological simulations of large-scale structure formation.
- Easy to incorporate periodic boundary conditions and expansion.

Hybrid Methods

Because PM method has low-resolution accuracy, it is often coupled with the direct N-body or hierarchical tree method to achieve larger range of applicability.

basic principle: Separate potential into near-field and far-field parts

$$\Phi(\mathbf{r}) = \Phi^{far}(\mathbf{r}) + \Phi^{near}(\mathbf{r})$$

In Fourier space,

$$\tilde{\Phi}(\mathbf{k}) = \tilde{\Phi}^{far}(\mathbf{k}) + \tilde{\Phi}^{near}(\mathbf{k})$$

$$-\frac{4\pi G\tilde{\rho}}{k^2} = -\frac{4\pi G\tilde{\rho}}{k^2}\exp\left(k^2 r_s^2\right) - \frac{4\pi G\tilde{\rho}}{k^2}\left[1 - \exp\left(k^2 r_s^2\right)\right]$$

Hybrid Methods

r_s : distance scale which separates "far" from "near".

The inverse Fourier transform of $\tilde{\Phi}^{near}$ gives

$$\Phi^{near}(\mathbf{r}) = -\frac{G\rho}{r}\text{erfc}\left(-\frac{r}{2r_s}\right)$$

complementary error function

$$\text{erfc}(x) \equiv 1\text{-erf}(x) = 1 - \frac{2}{\sqrt{\pi}}\int_0^x e^{-\tau^2}d\tau = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-\tau^2}d\tau$$

Hybrid Methods

The near-field acceleration is then

$$\mathbf{g}^{near}(\mathbf{r}) = -\frac{G\rho\mathbf{r}}{r^3}\left[\text{erfc}\left(-\frac{r}{2r_s}\right) + \frac{r}{r_s\sqrt{\pi}}\exp\left(-\frac{r^2}{4r_s^2}\right)\right]$$

This is calculated in real space using the Tree method or direct N-body method. The result is added to the far-field result from the PM method.

- When tree method is used, this is called the TreePM method
- When direct N-body is used, this is called the particle-particle-particle mesh method (P^3M)

Restricted N-body

The first groundbreaking galaxy simulation research used none of these methods.

Toomre & Toomre, 1972 used the restricted N-body method:

- Two massive particles interact gravitationally.
- All other particles are massless "tracer" particles orbiting in the gravitational field of the two massive particles.
- Very fast! Scales as N.
- Very limited realism, but still useful.

Time Integration

Now that we know how the forces / accelerations are calculated, we need to know how to advance the particles forward in time.

First, let's review the general procedure for calculating derivatives and error terms numerically:

1. Write Taylor expansions of the function $f(t)$ itself and shifted expressions such as $f\left(t \pm \delta t\right)$, $f\left(t \pm 2\delta t\right)$, $f\left(t \pm \frac{1}{2}\delta t\right), \cdots$ where δt is the step size.
2. Form linear combination of Taylor expansions to isolate the desired derivative with the desired order of accuracy.

Example 1: Centered Difference $f'(t)$

$$f\left(t+\delta t\right)=f\left(t\right)+\delta t f'\left(t\right)+\frac{1}{2}\delta t^2 f''\left(t\right)+\frac{1}{6}\delta t^3 f^{(3)}\left(\tau^+\right)$$

$$f\left(t-\delta t\right)=f\left(t\right)-\delta t f'\left(t\right)+\frac{1}{2}\delta t^2 f''\left(t\right)-\frac{1}{6}\delta t^3 f^{(3)}\left(\tau^-\right)$$

where $\tau^{\pm}$ is between t and $t\pm\delta t$.

Solving for $f'\left(t\right)$,

$$f'\left(t\right)=\frac{f\left(t+\delta t\right)-f\left(t-\delta t\right)}{2\delta t}-\frac{1}{6}\delta t^3 f^{(3)}\left(\tau\right)$$

where $t-\delta t\leq\tau\leq t+\delta t$.

Example 2: Centered Difference $f''(t)$

$$f\left(t+\delta t\right)=f\left(t\right)+\delta t f'\left(t\right)+\frac{1}{2}\delta t^{2}f''\left(t\right)+\frac{1}{6}\delta t^{3}f^{(3)}\left(\tau^{+}\right)$$

$$f\left(t-\delta t\right)=f\left(t\right)-\delta t f'\left(t\right)+\frac{1}{2}\delta t^{2}f''\left(t\right)-\frac{1}{6}\delta t^{3}f^{(3)}\left(\tau^{-}\right)$$

solving for $f''\left(t\right)$,

$$f''\left(t\right)=\frac{f\left(t+\delta t\right)-2f(t)+f\left(t-\delta t\right)}{\delta t^{2}}-\frac{1}{12}\delta t^{2}f^{(4)}\left(\tau\right)$$

where $\tau^{\pm}$ is between t and $t\pm\delta t$, thus $t-\delta t\leq\tau\leq t+\delta t$.

Example 3: $\mathcal{O}(\delta t^4)$ error $f'(t)$

$$f\left(t \pm \delta t\right) = f \pm \delta t f' + \frac{1}{2}\delta t^2 f'' \pm \frac{1}{6}\delta t^3 f^{(3)} + \frac{1}{24}\delta t^4 f^{(4)}$$

$$f\left(t \pm 2\delta t\right) = f \pm 2\delta t f' + 2\delta t^2 f'' \pm \frac{8}{6}\delta t^3 f^{(3)} + \frac{16}{24}\delta t^4 f^{(4)}$$

solving for $f'\left(t\right)$ while seeking $\mathcal{O}(\delta t^4)$ error,

$$f' = \frac{f\left(t + 2\delta t\right) + 8\left[f\left(t + \delta t\right) - f\left(t - \delta t\right)\right] - f\left(t - 2\delta t\right)}{12\delta t} + \mathcal{O}(\delta t^4)$$

Global Error

Error accumulates with each iteration. The accumulation is called global error.

$$\text{global error} \propto N_{iter} \times \text{local error}$$

$$= \frac{T}{\delta t}\mathcal{O}\left(\delta t^n\right) = T\mathcal{O}\left(\delta t^{n-1}\right)$$

- a method which is of order n per step is of order $n-1$ when iterated over time.
- Knowing this helps us to decide on an appropriate time-step.

Euler Method

Steps:

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_n \delta t$$

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \mathbf{v}_n \delta t$$

- First-order globally, second order locally
- poorly behaved
- simplest method

Midpoint Method

Steps:

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_n \delta t$$

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \frac{\mathbf{v}_n + \mathbf{v}_{n+1}}{2} \delta t$$

The net result:

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \mathbf{v}_n \delta t + \frac{1}{2} \mathbf{a}_n \delta t^2$$

- Second-order globally
- Velocity step is only first order
- This is an example of a second order Runge-Kutta method.

Runge-Kutta 4 (RK4)

First, the compact form of the RK4 algorithm:

- $\mathbf{x}$ is a vector of quantities being integrated.
- $\mathbf{f}$ is the derivative of $\mathbf{x}$ with respect to the parameter t (not necessarily time)

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}$$

The algorithm is

$$\mathbf{x}\left(t + \delta t\right) = \mathbf{x}\left(t\right) + \frac{1}{6}\delta t\left(\mathbf{F}_1 + 2\mathbf{F}_2 + 2\mathbf{F}_3 + \mathbf{F}_4\right)$$

...

Runge-Kutta 4 (RK4)

$$\mathbf{x}\left(t+\delta t\right)=\mathbf{x}\left(t\right)+\frac{1}{6}\left(\mathbf{F}_1+2\mathbf{F}_2+2\mathbf{F}_3+\mathbf{F}_4\right)\delta t$$

where

$$\begin{aligned}\mathbf{F}_1&=\mathbf{f}\left(\mathbf{x},t\right)\\ \mathbf{F}_2&=\mathbf{f}\left(\mathbf{x}+\frac{\delta t}{2}\mathbf{F}_1,t+\frac{1}{2}\delta t\right)\\ \mathbf{F}_3&=\mathbf{f}\left(\mathbf{x}+\frac{\delta t}{2}\mathbf{F}_2,t+\frac{1}{2}\delta t\right)\\ \mathbf{F}_4&=\mathbf{f}\left(\mathbf{x}+\delta t\mathbf{F}_3,t+\delta t\right)\end{aligned}$$

Runge-Kutta 4 (RK4)

written explicitly:

$$\mathbf{a}^{(1)} = \mathbf{a}_i = \mathbf{a}(r_i) \qquad \mathbf{v}^{(1)} = \mathbf{v}_i$$
$$\mathbf{a}^{(2)} = \mathbf{a}\left(\mathbf{r}_i + \mathbf{v}^{(1)}\delta t/2\right) \qquad \mathbf{v}^{(2)} = \mathbf{v}_i + \mathbf{a}^{(1)}\delta t/2$$
$$\mathbf{a}^{(3)} = \mathbf{a}\left(\mathbf{r}_i + \mathbf{v}^{(2)}\delta t/2\right) \qquad \mathbf{v}^{(3)} = \mathbf{v}_i + \mathbf{a}^{(2)}\delta t/2$$
$$\mathbf{a}^{(4)} = \mathbf{a}\left(\mathbf{r}_i + \mathbf{v}^{(3)}\delta t\right) \qquad \mathbf{v}^{(4)} = \mathbf{v}_i + \mathbf{a}^{(3)}\delta t$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \frac{1}{6}\left(\mathbf{a}^{(1)} + 2\mathbf{a}^{(2)} + 2\mathbf{a}^{(3)} + \mathbf{a}^{(4)}\right)\delta t$$

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \frac{1}{6}\left(\mathbf{v}^{(1)} + 2\mathbf{v}^{(2)} + 2\mathbf{v}^{(3)} + \mathbf{v}^{(4)}\right)\delta t$$

Runge-Kutta 4 (RK4)

Comments:

- 4th order accuracy for relatively little effort.
- By far, the most commonly used ODE solver for general-purpose use.
- NOT used for calculating orbits!
  - Does not conserve energy when calculating orbits.
  - Requires 4 acceleration evaluations per time-step!

Leap-Frog Method

Using the centered difference, compute acceleration at time t,

$$\mathbf{a}\left(\mathbf{r}\left(t\right)\right) = \frac{\mathbf{v}\left(t+\delta t\right) - \mathbf{v}\left(t-\delta t\right)}{2\delta t} + \mathcal{O}\left(\delta t^2\right)$$

and the velocity at time $t + \delta t$,

$$\mathbf{v}\left(t+\delta t\right) = \frac{\mathbf{r}\left(t+2\delta t\right) - \mathbf{r}\left(t\right)}{2\delta t} + \mathcal{O}\left(\delta t^2\right)$$

Then rearrange terms so that future values are on the left

$$\mathbf{v}\left(t+\delta t\right) = \mathbf{v}\left(t-\delta t\right) + 2\mathbf{a}\left(\mathbf{r}\left(t\right)\right)\delta t + \mathcal{O}\left(\delta t^3\right)$$

$$\mathbf{r}\left(t+2\delta t\right) = \mathbf{r}\left(t\right) + 2\mathbf{v}\left(t+\delta t\right)\delta t + \mathcal{O}\left(\delta t^3\right)$$

Leap-Frog Method

Using index notation,

$$\mathbf{v}_{i+1} = \mathbf{v}_{i-1} + 2\mathbf{a}_i \delta t + \mathcal{O}\left(\delta t^3\right)$$

$$\mathbf{r}_{i+2} = \mathbf{r}_i + 2\mathbf{v}_{i+1} \delta t + \mathcal{O}\left(\delta t^3\right)$$

This is sometimes written as

$$\mathbf{v}_{i+\frac{1}{2}} = \mathbf{v}_{i-\frac{1}{2}} + \mathbf{a}_i \delta t + \mathcal{O}\left(\delta t^3\right)$$

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_{i+\frac{1}{2}} \delta t + \mathcal{O}\left(\delta t^3\right)$$

and other ways...

Leap-Frog Method

drift-kick-drift version:

$$\mathbf{r}_{i+\frac{1}{2}} = \mathbf{r}_i + \mathbf{a}_i \frac{\delta t}{2}$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_{i+\frac{1}{2}} \delta t$$

$$\mathbf{r}_{i+1} = \mathbf{r}_{i+\frac{1}{2}} + \mathbf{v}_{i+1} \frac{\delta t}{2}$$

where

$$\mathbf{a}_{i+\frac{1}{2}} = \mathbf{a}\left(\mathbf{r}_{i+\frac{1}{2}}\right)$$

kick-drift-kick version:

$$\mathbf{v}_{i+\frac{1}{2}} = \mathbf{v}_i + \mathbf{a}_i \frac{\delta t}{2}$$

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_{i+\frac{1}{2}} \delta t$$

$$\mathbf{v}_{i+1} = \mathbf{v}_{i+\frac{1}{2}} + \mathbf{a}_{i+1} \frac{\delta t}{2}$$

where

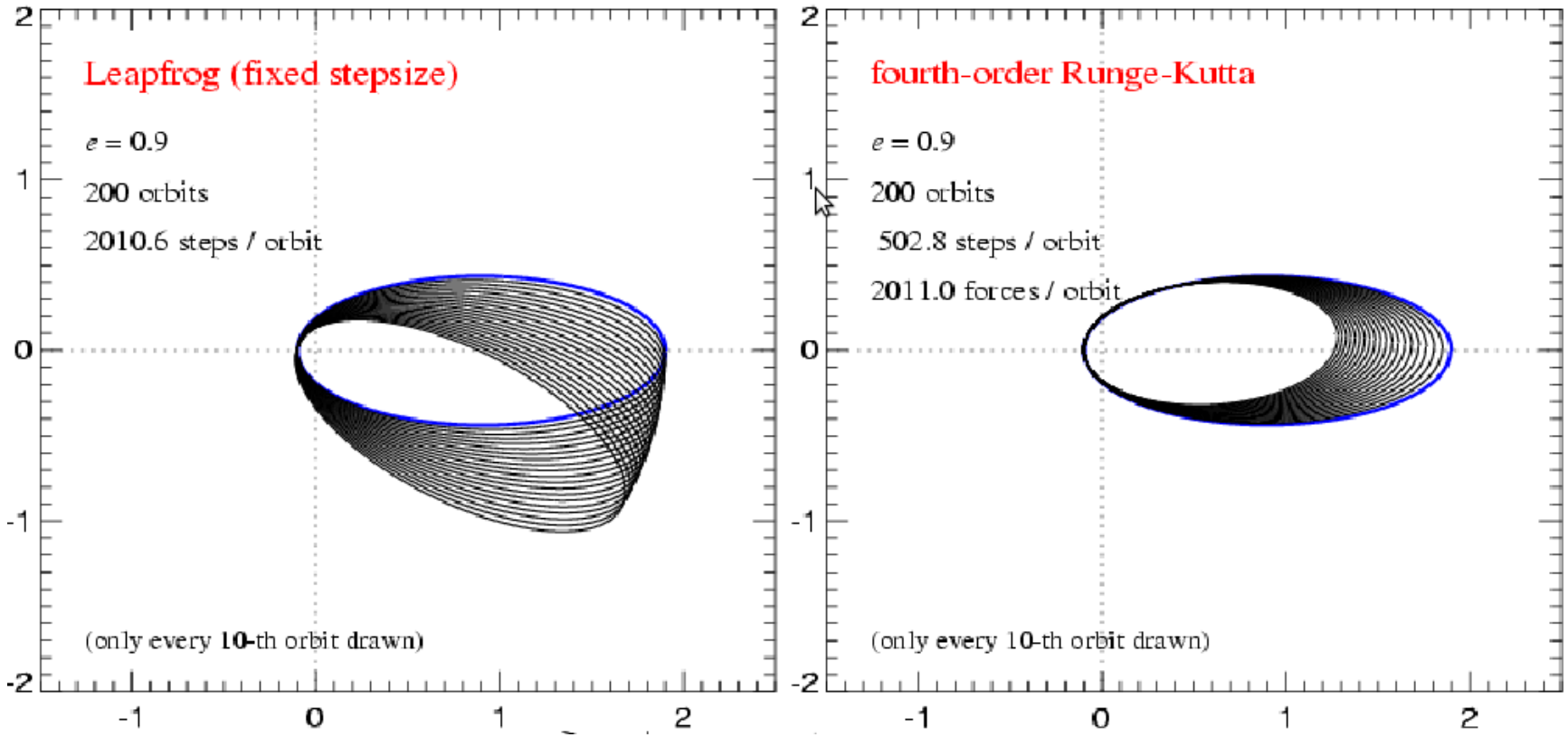
$$\mathbf{a}_{i+1} = \mathbf{a}\left(\mathbf{r}_{i+1}\right)$$

Leap-Frog Method

Comments:

- Even though it is only second order, Leap-Frog yields better results than RK4!
- Leap-frog method **conserves energy** because it is symmetric in time (It is *symplectic*. See Springel notes for more details)
- Kick-drift-kick form is more stable than drift-kick-drift when using *variable* step sizes; it is the most symmetric form in that case.

Comparison of Methods



Choosing a Step Size

In practice, adaptive step sizes must be used to assure desired accuracy

- In general, larger accelerations require smaller step sizes.
- One acceleration-based choice:

$$\delta t = f_{s,a}\sqrt{\frac{\epsilon}{|\mathbf{a}|}}$$

  where $f_{s,a} < 1$ is a "safety factor" determined from numerical experimentation.
- It's also a good idea to keep particles from moving too far during one time step...

Choosing a Step Size

An upper-limit on step size:

$$\delta t = f_{s,v} \frac{\epsilon}{|\mathbf{v}|} = \frac{d_{max}}{|\mathbf{v}|}$$

Where $f_{s,v}$ is another experimentally-determined constant factor. This keeps the particle from drifting farther than d_{max} during one time step.

But how are $f_{s,a}$ and $f_{s,v}$ experimentally chosen? ...

Choosing a Step Size

The accuracy of the integration code can be tested in several ways:

- by comparison with analytic solutions of orbital trajectories (for simple 2 particle test systems)
- by comparison with very high resolution simulation (in the limit as $\delta t \rightarrow 0$, the solution is exact—within machine precision)
- by testing for conservation of energy, linear momentum, and angular momentum

Based on the results from these tests, one can determine values of $f_{s,a}$ and/or $f_{s,v}$ which yield the desired solution accuracy.

Adaptive step sizes lead to a new problem in N-body simulations: synchronization...

Synchronization

- There is a large range of acceleration values in an N-body code.
- This implies a large range of optimal step-sizes.
- System must remain synchronized.

problem:

- The simple solution is to use the minimum time step globally.
  - if *one* particle in the simulation experiences a large acceleration, *all* other particles are integrated with small times.
  - VERY inefficient!

Synchronization

Solution:

- Create hierarchy of step sizes.
- common choice: choose step sizes as a power of 2 times the smallest required step size.
- Evolve highest-acceleration / smallest step size particles forward for, 2^n iterations, then evolve next-smallest time steps and so on until the largest step sizes are advanced.
- Then repeat

Outline

Fluid Dynamics & Other Physics

- Galaxies are more than systems of gravitationally-bound stars. Other constituents are very important.
  - gas, plasma, dust
  - magnetic fields, EM radiation
- Stars evolve with time. Simulations should include
  - Stellar formation
  - Stellar mass loss/gain
  - Stellar death

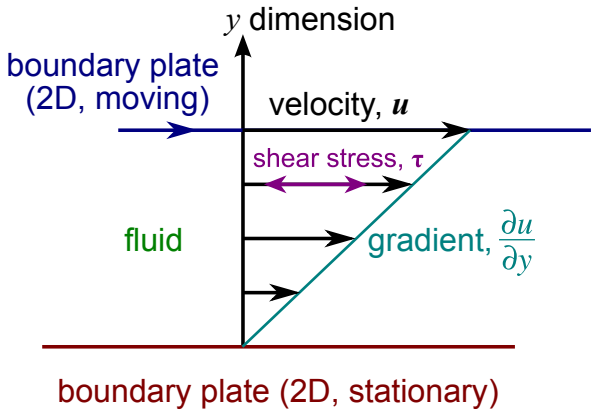
Fluid Quantities

ρ: mass density
n: particle number density
$\mathbf{v}$: velocity
$\rho\mathbf{v}$: momentum density
s: specific entropy
$\varepsilon = C_v T$: specific energy density
$E = \rho\varepsilon + \frac{1}{2}\rho v^2$: total energy density

V : volume
T : temperature
p : pressure
C_v: specific heat, constant V
C_p: specific heat, constant p
$\gamma = C_p/C_v$: adiabatic index
$c_s = \sqrt{\gamma p/\rho}$: speed of sound
μ: dynamic viscosity
$\nu = \mu/\rho$: kinematic viscosity

Viscosity

- Viscosity is a measure of fluid friction.
- stress: force per unit area.
- For shear stress, τ, surface is parallel to force.
- Assume: fluid is stationary with respect to surface.



$$\tau = \mu \frac{du}{dy}$$

Fluid Shocks

Shock: A very thin region over which fluid pressure, velocity (and/or temperature) change significantly.

- Often associated with (caused by) super-sonic flow.
- If fluid is viscous, shocks have finite width.
- For inviscid fluids, shocks can be modeled analytically as infinitesimally thin sheets / boundaries.
- Shocks are irreversible in time: entropy increases.

Examples: Sonic "boom", Thunder, Blast fronts (from explosions), bow shock, collisions of molecular clouds.

Contact Discontinuity

Contact Discontinuity: A very thin region over which fluid density and temperature change significantly (essentially discontinuously).

- Pressure is continuous across a contact discontinuity.
- There is no particle transport across (normal to) the discontinuity.
- If the tangential velocity is discontinuous, it is classified as a **slip** discontinuity.

Examples: The boundary between a supersonic jet and surrounding air (rocket engine exhaust), a planetary magnetopause

Fluid Equations

Navier-Stokes:

- Best available description of un-charged fluids.
- Includes conservation of mass, energy & momentum.
- Includes frictional forces (viscosity).

Magnetohydrodynamics (MHD):

- Couples Navier-Stokes with Maxwell equations
- Describes plasmas and other conducting fluids (e.g. liquid metals, ocean water).

Euler:

- Similar to Navier-Stokes, but for inviscid fluids.
- Conserves mass, energy, & momentum.

Review: Continuity / Conservation

Differential form of the continuity equation:

$$\frac{\partial f}{\partial t} + \nabla \cdot f\mathbf{v} = \sigma$$

Where σ is a source/sink term. $\sigma = 0$ implies the *conservation* of the quantity f.

The integral form:

$$\frac{\partial}{\partial t}\int_V f dV + \int_V \nabla \cdot f\mathbf{v} dV = \Sigma$$

$$\frac{\partial}{\partial t}\int_V f dV + \int_S f\mathbf{v} \cdot d\mathbf{S} = \Sigma$$

where $\Sigma = \int_V \sigma dV$.

The Euler Equations

Conservation of mass:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0$$

Conservation of momentum:

$$\frac{\partial \rho \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla) \rho \mathbf{v} = \nabla \cdot \bar{\sigma}$$

Where $\bar{\sigma}$ is the stress tensor. Assuming there are no shear stresses (i.e. no viscosity) and no body forces,

$$\left(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla \right) \rho \mathbf{v} = -\nabla p$$

The "convective derivative" or "material derivative" is defined as

$$\frac{D}{Dt} \equiv \frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla$$

Using this, momentum conservation is

$$\frac{D\rho\mathbf{v}}{Dt} + \nabla p = 0$$

This is simply $\mathbf{F} = m\mathbf{a}$

Conservation of energy:

$$\frac{\partial E}{\partial t} + \nabla \cdot (E + p)\,\mathbf{v} = 0$$

Computation-ready Euler Equations

$$\frac{\partial \mathbf{m}}{\partial t} + \frac{\partial \mathbf{f}_x}{\partial x} + \frac{\partial \mathbf{f}_y}{\partial y} + \frac{\partial \mathbf{f}_z}{\partial z} = 0$$

$$\mathbf{m} = \begin{bmatrix} \rho \\ \rho\dot{x} \\ \rho\dot{y} \\ \rho\dot{z} \\ E \end{bmatrix}, \mathbf{f}_x = \begin{bmatrix} \rho\dot{x} \\ p + \rho\dot{x}^2 \\ \rho\dot{x}\dot{y} \\ \rho\dot{x}\dot{z} \\ (E+p)\,\dot{x} \end{bmatrix}, \mathbf{f}_y = \begin{bmatrix} \rho\dot{y} \\ \rho\dot{x}\dot{y} \\ p + \rho\dot{y}^2 \\ \rho\dot{y}\dot{z} \\ (E+p)\,\dot{y} \end{bmatrix}, \mathbf{f}_z = \begin{bmatrix} \rho\dot{z} \\ \rho\dot{x}\dot{z} \\ \rho\dot{y}\dot{z} \\ p + \rho\dot{z}^2 \\ (E+p)\,\dot{z} \end{bmatrix}$$

close the equations with

$$p = \rho\varepsilon\,(\gamma - 1)$$

Mesh-based Eulerian Methods

Some vocabulary

Mesh-based method: Uses a mesh / grid to discritize the fluid.

Eulerian method: Fluid quantities are stored at fixed points. Fluid passes by the points.

Lagrangian method: Fluid quantities are stored at points which flow *with* the fluid.

We will briefly overview the following mesh-based, Eulerian methods:

- Finite Difference (FDM)
- Finite Volume (FVM)
- Finite Element (FEM)

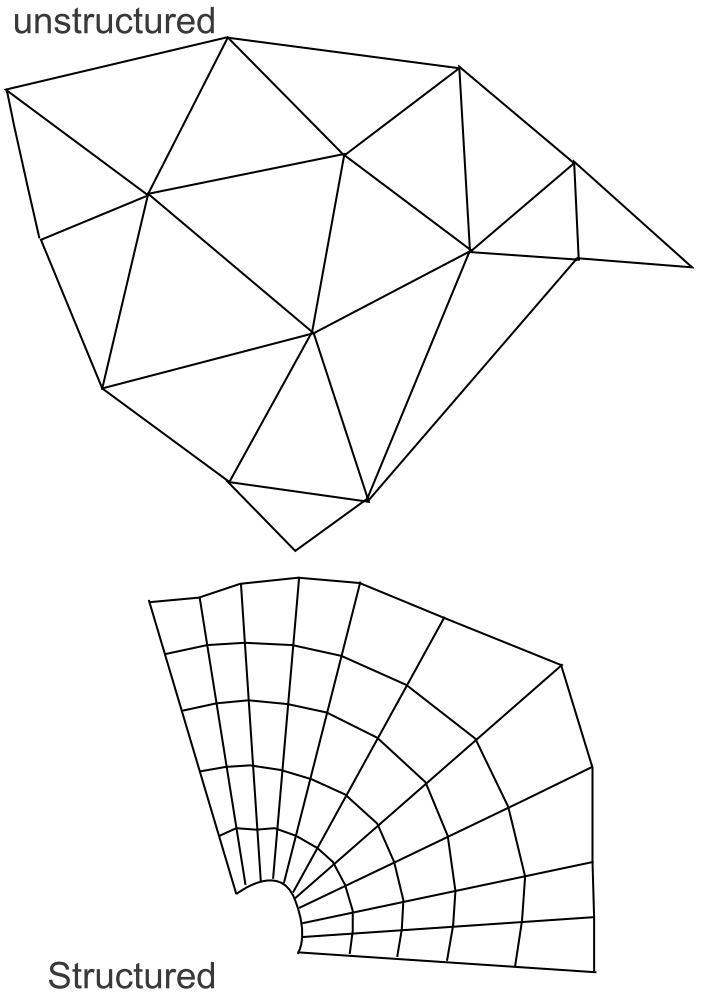
Finite Difference Method

In general terms, FDM methods use finite differencing to calculate derivatives on structured meshes.

- Structured meshes have a regular connectivity pattern. For FDM, the meshes are generally orthogonal or warped orthogonal grids.
- Continuum quantities are usually stored at the nodes of the mesh–sometimes the cell centers.
- Linear algebra (and other) routines are used to solve the PDEs.
- FDM is restrictive because the mesh has to be structured.

Unstructured Meshes

- More versatile.
- There is no fixed connectivity pattern.
- Connectivity information is stored in data structures.
- Mesh cells can be added / removed as needed.



Finite Volume Method

If we write the equations in explicitly conservative integral form, the fluid equations are solved easily. In particular, the integral form of the Euler equations:

$$\frac{\partial}{\partial t}\int_V \mathbf{m}\, dV = -\int_S \mathbf{f}\cdot d\mathbf{S}$$

- Divide space into finite volumes using unstructured mesh.
- Store fluid quantities in centers of volumes.
- Assume the quantities are piecewise constant.
- The integral equation above is approximated by

$$\sum_j \frac{\partial}{\partial t}\mathbf{m}_j\, V_j = -\sum_i \mathbf{f}_{ij}\cdot \mathbf{S}_i$$

Finite Volume Method

$$\sum_j \frac{\partial}{\partial t} \mathbf{m}_j V_j = -\sum_i \mathbf{f}_{ij} \cdot \mathbf{S}_{ij}$$

- $\mathbf{S}_{ij}$ is the area of the ith face of the jth volume element.
- $\mathbf{f}_{ij}$ is the vector of flux densities (calculated as the average of $\mathbf{f}$ on either side of the face i, j)
- Use Runge-Kutta to solve the differential equations.
- This procedure is only accurate to first order.
- Higher order accuracy can be obtained using a trick...

Finite Volume Method

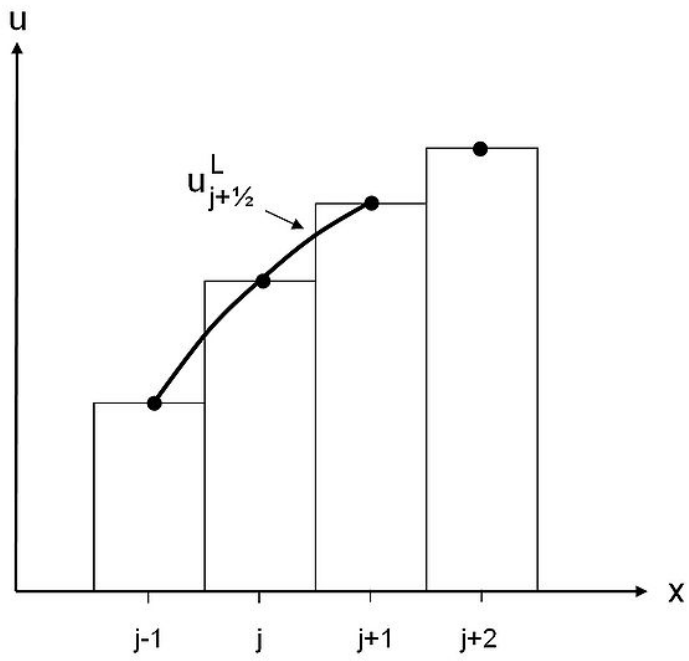
Riemann solver:

- The Riemann problem consists of two piecewise constant fluids in contact.
- This is identical to the case of the interface between finite volume cells.
- There is an exact, analytic solution to the Riemann problem.
- Exact solution can be used to determine how the fluid evolves with time.
- Implementing this in FVM leads to first-order accuracy.
- But, we can improve this...

Finite Volume Method

Piecewise Parabolic Method (PPM):

- Reconstruct fluid variables in each volume using quadratic interpolation from surrounding volumes.
- Calculate fluxes at boundaries using interpolation.
- Fluxes at edges are input into a Riemann solver.



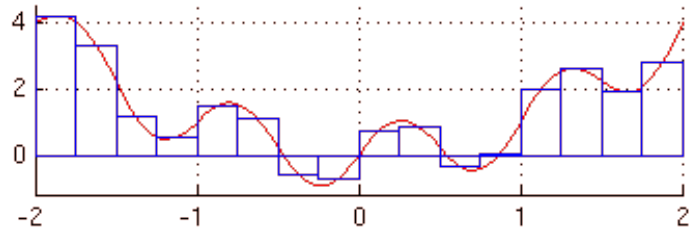
- Third order accuracy!

Finite Element Method

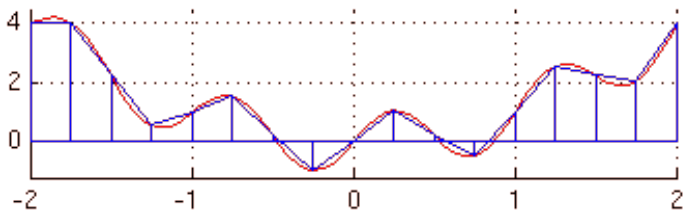
Rather than storing fluid quantities as piecewise constant, use interpolating functions in each cell.

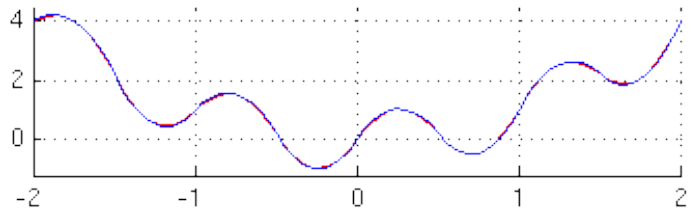
Analogy with integration methods:

- FVM is analogous to midpoint rule for numerical integration:



- FEM is analogous to trapezoid rule or Simpson's rule:





Finite Element Method

- Quantities are interpolated in each element using the expansion

$$f(\mathbf{r}) \approx \sum_k f_k w_k(\mathbf{r})$$

- f_k are weights, $w_k(\mathbf{r})$ are basis functions or "shape functions".
- Using the expansion, the system of PDEs is approximated by a system of ODEs.
- Solve using linear algebra routines for systems of ODEs.
- Higher order shape functions lead to higher order accuracy.
- Second and third order solutions are relatively easy to obtain.
- Standard method for state of the art engineering codes.

Limitations

Problems with mesh-based Eulerian methods:

- Interfaces between different fluids (e.g. liquid-gas) are difficult.
- Gas-vacuum boundaries are inefficiently calculated.
- Adaptive mesh refinement required for large dynamic range.
- Methods are not Galilean-invariant, thus flows are often supersonic through mesh.
- Over-mixing causes spurious entropy creation.
- Self-gravitating fluids are difficult to model.

Lagrangian Methods

The main idea: Sample fluid properties using points which move with the bulk flow of the fluid.

- Galilean invariance is achieved.
- Self-gravitation can be included more easily.

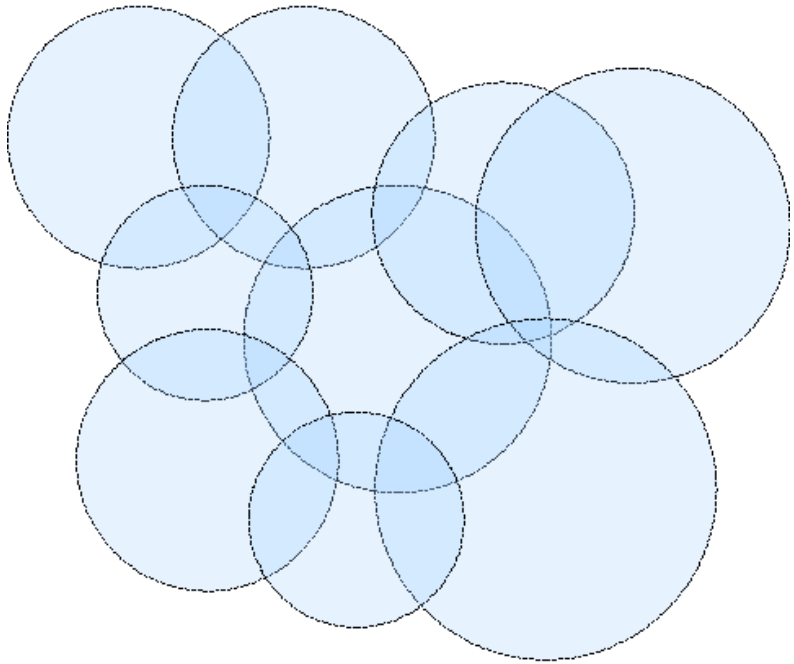
We will discuss two approaches:

- Meshless method: Smoothed-Particle Hydrodynamics (SPH).
- Mesh-based methods: The moving mesh

SPH

Basic principle:

- Fluid quantities are sampled at discrete points with fixed mass.
- Particles move with the bulk flow of the fluid.
- Particles are *smoothed* over spherical regions of radius h, called "smoothing length".
- Particles are required to overlap.



- Specify minimum number of required overlaps, then scale h.

Basic Formulation

Start with the trivial identity:

$$A(\mathbf{r}) = \int A(\mathbf{r}')\delta\left(\mathbf{r} - \mathbf{r}'\right) d\mathbf{r}'$$

This can be approximated

$$\langle A(\mathbf{r})\rangle \approx \int A(\mathbf{r}')w\left(\mathbf{r} - \mathbf{r}', h\right) d\mathbf{r}'$$

where the smoothing kernel $w(r, h)$ obeys

$$\int w(r, h)dV = 1$$

and

$$\lim_{h\to 0} w(r, h) \longrightarrow \delta(r)$$

$$\langle A(\mathbf{r})\rangle \approx \int A(\mathbf{r}')w\left(\mathbf{r}-\mathbf{r}',h\right)d\mathbf{r}'$$

$$= \int \frac{A(\mathbf{r}')}{\rho\left(\mathbf{r}'\right)}\rho\left(\mathbf{r}'\right)w\left(\mathbf{r}-\mathbf{r}',h\right)d\mathbf{r}'$$

This is approximated as

$$\langle A(\mathbf{r}_i)\rangle \approx \sum_j \left[\rho\left(\mathbf{r}_j\right)V_j\right]\frac{A\left(\mathbf{r}_j\right)}{\rho\left(\mathbf{r}_j\right)}w\left(\mathbf{r}_i-\mathbf{r}_j,h\right)$$

$$= \sum_j m_j\frac{A_j}{\rho_j}w\left(\mathbf{r}_i-\mathbf{r}_j,h\right)$$

We arrive at the approximation

$$\langle A(\mathbf{r}_i)\rangle \approx \sum_j m_j \frac{A_j}{\rho_j} w\left(\mathbf{r}_i - \mathbf{r}_j, h\right)$$

this is analogous to FEM expansion in terms of shape functions.

Example: calculate $\langle \rho(\mathbf{r}_i)\rangle$

$$\langle \rho(\mathbf{r}_i)\rangle \approx \sum_j m_j \frac{\rho_j}{\rho_j} w\left(\mathbf{r}_i - \mathbf{r}_j, h\right) = \sum_j m_j w\left(\mathbf{r}_i - \mathbf{r}_j, h\right)$$

Example: calculate $\langle \mathbf{v}_i \rangle$

$$\langle \mathbf{v}_i \rangle \approx \sum_j \frac{m_j}{\rho_j} \mathbf{v}_j w\left(\mathbf{r}_i - \mathbf{r}_j, h\right)$$

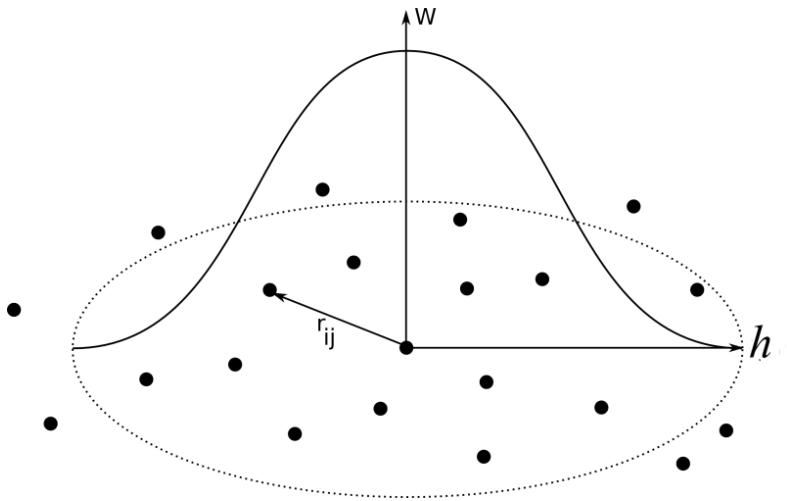
Gradient and divergence are approximated by returning to integral formulation, using integration by parts and vector identities to manipulate the expressions. The results are:

$$\nabla A(\mathbf{r}_i) \approx \langle \nabla A(\mathbf{r}_i) \rangle \approx \sum_j m_j \frac{A_j}{\rho_j} \nabla_i w\left(\mathbf{r}_i - \mathbf{r}_j, h\right)$$

$$\nabla \cdot \mathbf{v}_i \approx \frac{1}{\rho_i} \sum_j \frac{m_j}{\rho_j} \left(\mathbf{v}_i - \mathbf{v}_j\right) \cdot \nabla_i w\left(\mathbf{r}_i - \mathbf{r}_j, h\right)$$

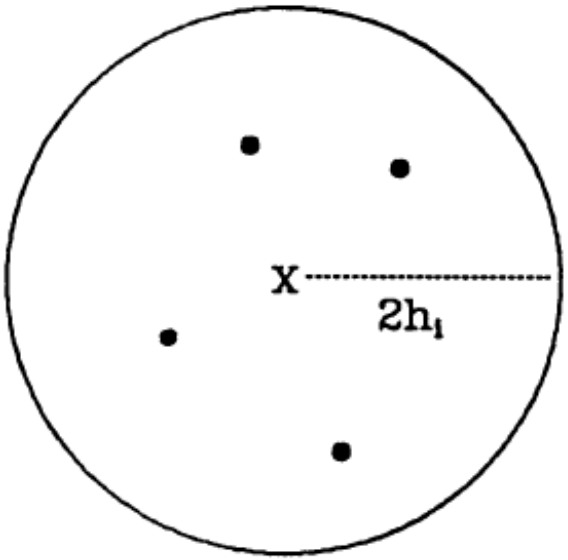
Gather vs. Scatter

- Smoothing kernels must overlap.
- Require minimum number of overlapping neighbors.
- This implies that the smoothing length can vary.
- There are two ways to interpret SPH: Gather and Scatter.

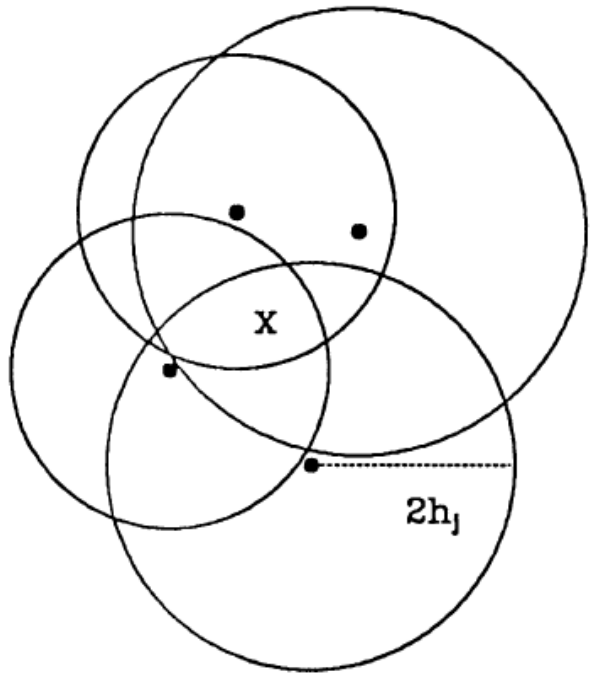


- Gather and scatter interpretations lead to different scaling condition for h

Gather vs. Scatter



Gather



Scatter

$$A(\mathbf{r}_i)_g \approx \sum_j m_j \frac{A_j}{\rho_j} w\left(\mathbf{r}_i - \mathbf{r}_j, h_i\right) \qquad A(\mathbf{r}_i)_s \approx \sum_j m_j \frac{A_j}{\rho_j} w\left(\mathbf{r}_i - \mathbf{r}_j, h_j\right)$$

Symmetrized version

- Gather and scatter are both third-order methods.
- The symmetrized version gives somewhat better results:

$$A(\mathbf{r}_i)_{sym} \approx \sum_j m_j \frac{A_j}{\rho_j} \frac{1}{2} \left[w\left(\mathbf{r}_i - \mathbf{r}_j, h_j\right) + w\left(\mathbf{r}_i - \mathbf{r}_j, h_i\right) \right]$$

- But still does not conserve energy and entropy simultaneously.
- For a fully-conservative adaptive smoothing scheme, see Springel notes.

Comments on SPH

Advantages:

- Galilean invariant
- Mass is automatically conserved.
- Handles fluid-fluid and fluid-vacuum interfaces naturally.
- Does not waste computational effort on voids/vacuum.

Disadvantages:

- Does not handle fluid shocks well
- Tends to dampen fluid instabilities†
- Relies heavily on artificial viscosity†

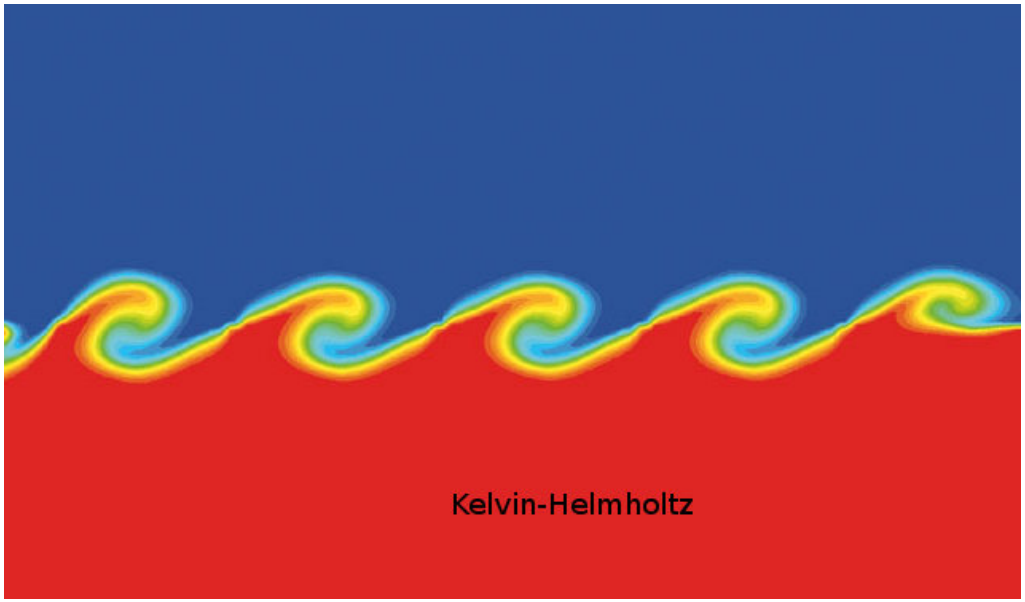
† see next four pages.

Fluid Instability

Non-linear effects sometimes associated with positive feedback loop.

Examples:

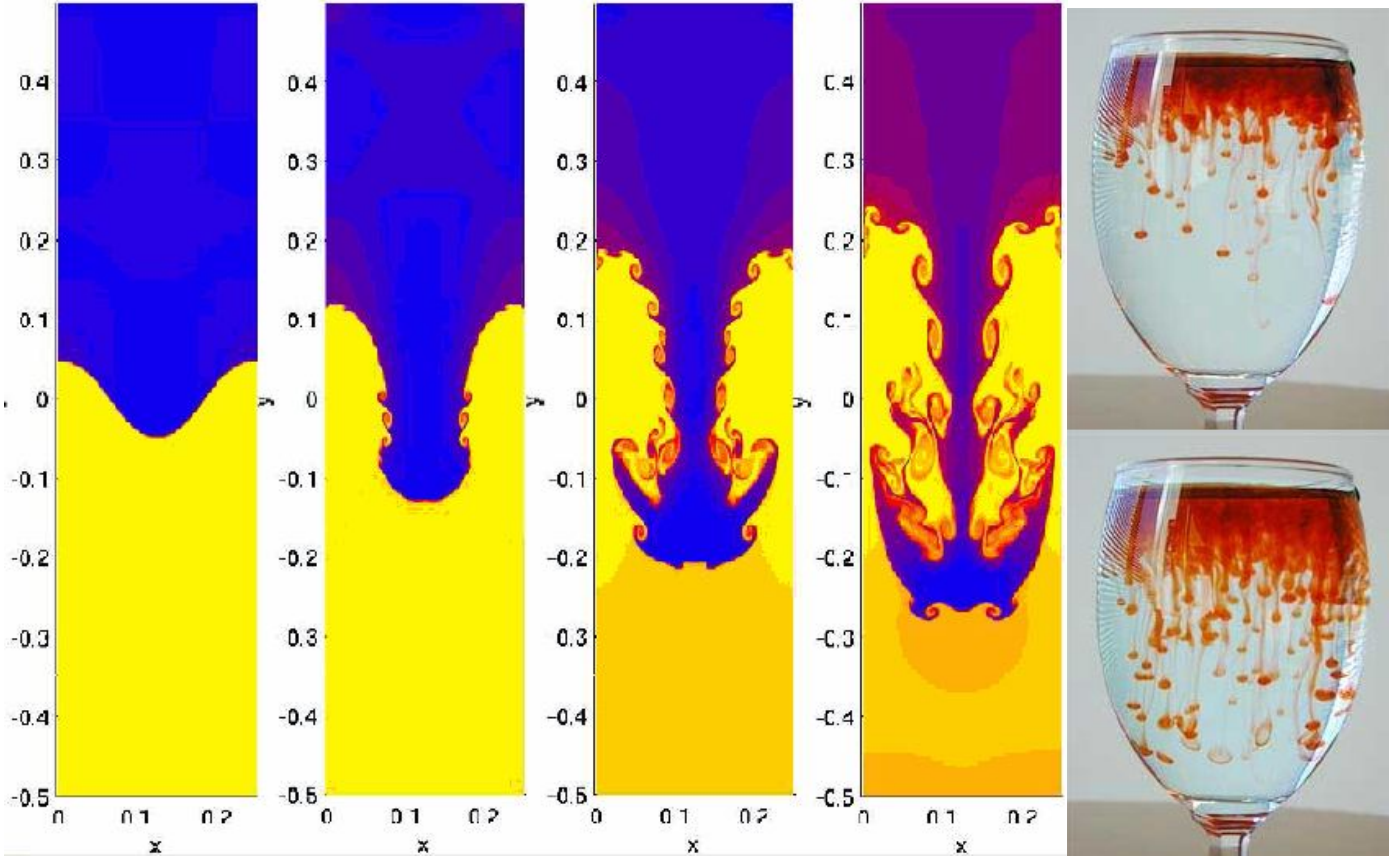
- Kelvin-Helmholtz instability
- Rayleigh-Taylor instability



Kelvin-Helmholtz Instability

Rayleigh-Taylor Instability

AKA "fingering instability": A low-density fluid pushes on a higher density fluid.



Artificial Viscosity

- Shocks in frictionless fluids can be arbitrarily thin. This can lead to unphysical singularities, overshoots, & oscillations.
- In real fluids, viscosity converts bulk kinetic energy into thermal energy at a shock.
- Real shocks have finite thickness...no singularities etc.
- Artificial viscosity terms are introduced to simulate the effects of real viscosity; Entropy is artificially added to simulate the actual increase of entropy at a shock.

Lagrangian Mesh

Idea: Combine the best features of mesh-based methods and SPH.

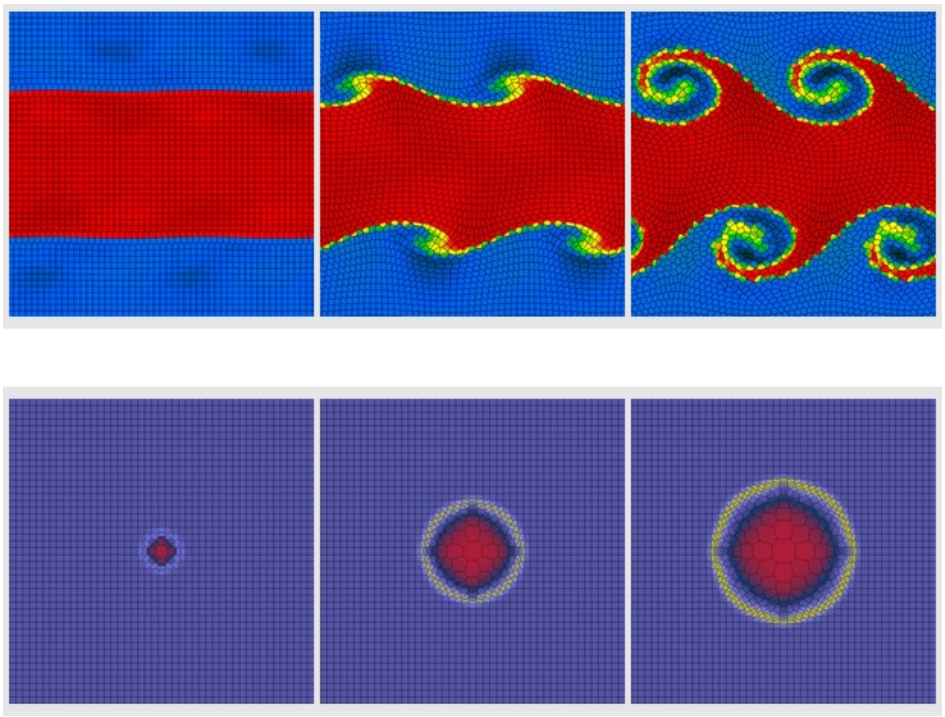
- Space is discretized using a mesh–no overlapping cells.
- Mesh elements move with fluid.
- Mesh elements can change size as fluid expands or contracts.
- Use finite volume method with Riemann solver for high-order accuracy.

Result:

- Code retains almost all of the advantages of SPH.
- Handles shocks/instabilities without need for artificial viscosity.

Lagrangian Mesh

Arepo: `http://www.mpa-garching.mpg.de/~volker/arepo`



Combining CFD & Gravity

We must incorporate gravitational forces & potential into CFD. Momentum & energy conservation need to be modified.

Conservation of mass: (Unchanged)

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0$$

Conservation of momentum:

$$\left(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla \right) \rho \mathbf{v} + \nabla p = -\rho \nabla \Phi$$

Conservation of energy:

$$\frac{\partial E}{\partial t} + \nabla \cdot (E + p) \mathbf{v} = -\rho \mathbf{v} \cdot \nabla \Phi$$

Combining CFD & Gravity

- Fluid, stellar, & dark matter particles interact as gravitationally-softened particles.
- Force and potential are calculated using N-body methods.
- Gravitational force / potential is included in CFD code.
- Particles in system are stepped forward in time.
- Fluid properties are stepped forward in time.

CFD Timestep Size

For a CFD simulation, the maximum timestep size is given by the Courant–Friedrichs–Lewy (CFL) condition.

$$\delta t_{max} = C_{CFL} \frac{L}{(v_{max} + c_s)}$$

δt_{max}: maximum allowed timestep.
L : Characteristic length (smallest dimension) of fluid element.
v_{max}: fastest possible bulk velocity within fluid element.
c_s: speed of sound.
C_{CFL}: Safety factor. $C_{CFL} < 1$.

Maximum Step Size

- For Lagrangian methods, v_{max} is negligible because volumes flow with bulk velocity of fluid.
- Thus another advantage of Lagrangian codes: larger timesteps.

When combining CFD with gravitation, step size is limited by smallest of the two steps:

$$\delta t_{max} = \min\left(\delta t_{grav}, \delta t_{CFL}\right)$$

Microphysics & Feedback

Other phenomena must be included:

- Gas & dust cooling + chemical reactions
- Magnetohydrodynamics
- Stellar formation / evolution / winds
- Supernova feedback
- AGN feedback

* Most of these are poorly understood.
*All are beyond the resolution of galaxy models.

Sub-resolution methods

Analytic approximations and empirical formulas are used to approximate the unresolvable physics. The details of the methods differ greatly, but in general...

- Cooling and coupling to EM radiation can be achieved by adding terms to energy and momentum equations.
- Stars can gain and lose mass with time. Mass is added / subtracted from gas particles and star particles.
- Stellar formation is based on empirical models (IMF) and fluid density and temperature conditions.
- SMBHs are treated as mass sinks.
- SMBH accretion rate is calculated using semi-analytic expression.
- AGN luminosity and SED is based on accretion rate and empirical relation respectively.

Sub-resolution methods

- SN explosions...
  - add mass source term to the fluid equations(ISM).
  - inject energy to the surrounding; add source of E equation.
  - transfer momentum to ISM; add source to $\mathbf{p}$ equation.
- Starbursts also heat and transfer momentum to ISM.
  - add source terms to E and $\mathbf{p}$.

Sample: Bondi-Hoyle-Lyttleton BH accretion rate

$$\dot{M} = \frac{4\pi\alpha G^2 M^2 \rho}{(c_s^2+v_\infty^2)^{3/2}} \quad \text{limited by } \dot{M}_{Edd}.$$

Initialization

There are several methods of initializing galaxies. Here's one method for spiral galaxies:

- Choose disk scale length, D, bulge scale length, b, dark matter halo scale size a, dark matter mass M_{dm}, stellar mass in disk, M_*, stellar mass in bulge, M_b, mass of gas M_g , disk scale height, z_0.
- Use the spherical dark matter density profile:

$$\rho_{dm}(r) = \frac{M_{dm}}{2\pi} \frac{a}{r(r+a)^3}$$

- Cylindrical stellar mass distribution in disk:

$$\rho_*(r,z) = \frac{M_*}{4\pi D^2} \exp(-r/D) \operatorname{sech}\left(\frac{z}{2z_0}\right)$$

Initialization

- Spherical distribution of stars in bulge:

$$\rho_b(r) = \frac{M_b}{2\pi} \frac{b}{r(r+b)^3}$$

- Cylindrical gas distribution in the disk:

$$\Sigma_g(r) = \frac{M_g}{2\pi D^2} \exp(-r/D)$$

- Determine vertical profile of gas in the disk from hydrostatic equilibrium:

$$\frac{1}{\rho_g}\frac{\partial p}{\partial z} + \frac{\partial \Phi}{\partial z} = 0$$

Initialization

- Choose mass / particle distribution for stellar particles.
- Solve the Jeans equations to find the kinematic structure for stable disk (see Binney & Tremaine).
- Sample the resulting distribution function randomly.

Other options:

- Reconstruct galaxy from observational data.
- Initialize galaxy using output of cosmological structure formation simulation.

Initialization

Initializing cosmological simulations: Several options are available. One popular method is called "making glass"...

- Begin with a randomly sampled Poisson distribution:

$$f(n, \lambda) = \frac{\lambda^n e^{-\lambda}}{n!}$$

- Run simulation with repulsive gravity and no feedback mechanisms.

Resulting distribution is reminiscent of particle distribution in glass.

Some Codes

GADGET-2: SPH, TreePM (monopole), public, feedback is not included by default.
GADGET-3: SPH, TreePM, not-public, full physics, more efficient code (cited as GADGET-2).
FLASH: AMR, PPM Riemann, MHD, PM (mostly for stellar-scale simulations).
Enzo: AMR, PPM Riemann, PM, radiative transfer, MHD, some chemistry. Public (UCSD).
Gasoline: SPH, Tree (hexadecapole), some extra physics is included (e.g. cooling).
Arepo: Lagrangian mesh, PPM Riemann, TreePM (monopole).
Athena: AMR, PPM Riemann, PM, detailed MHD.

Testing for accuracy & precision

Fluid codes are validated by:

- Comparing simulation results with standard analytic solutions: (e.g. shock tubes, Prandtl-Meyer expansion wave, laminar flow over a circular cylinder, Sedov-Taylor point explosion)
- Comparing simulation results from previously validated codes (on a problem that both codes are capable of solving)

To determine precision & accuracy of galaxy simulations,

- Take into account the fundamental accuracy of the code, based on the above methods.
- Run simulations at higher resolutions to characterize scaling with mesh size / smoothing (h), softening (ϵ), and N.
- Compare results with observations of real systems.

Testing for accuracy & precision

A note on shot noise / particle noise:

When measuring physical quantities such as velocity dispersion, mass density, and energy density in particle-based simulations, there is random noise in the measurement—just as is the case when observing objects with a finite number of photons.

- The "signal uncertainty", σ_q scales as

$$\sigma_q \propto 1/\sqrt{N}$$

- σ_q is the standard deviation of the quantity q which can be determined by making many measurements of q (from different directions, at different times, in different locations, etc., depending on the details of the simulation)

Outline

System Architecture

In order to implement the algorithms previously presented, one needs to know something about computers.

Five main components:

- Input
- Memory
- Communications system
- Processor
- Output

System Architecture

Input: Computer needs instructions from the outside world.

- Keyboard
- Cameras
- Microphones
- Disk
- Other computers (network)

System Architecture

Output: Computer needs to communicate results with the outside world.

- Monitors
- Speakers
- Printers
- Actuators
- Disk
- Other computers (network)

System Architecture

Memory: Computer needs to store data. Basic principle: larger storage is slower and less expensive.

HDD / SSD:

- Non-volatile
- least expensive per byte
- Slowest transfer rates ($\sim$10MB/s – 1000MB/s)
- Highest capacity ($\sim$TB)
- Highest latency ($\sim 10\mu$s - $\sim$1ms)

Memory

System memory: compared to HDD / SDD...

- More expensive per byte
- Lower density
- Smaller capacity ($\sim$GB)
- Volatile
- Faster transfer rates ($\sim$1GB/s – $\sim$10GB/s)
- Lower latency ($\sim$ns)
- Connected to processor more directly

Memory

Cache memory: compared to system memory...

- much more expensive per byte
- modern caches are integrated into the processor die
- 3 levels, in order of decreasing latency & capacity:
  - L3: largest (2-20MB), shared among several processors$*$.
  - L2: smaller (0.5-2MB) dedicated to one processor.
  - L1: smallest (0.25-1MB) very close to execution units.†
- Very high transfer rates ($\sim$100GB/s – 1TB/s).

$\dagger$ Subdivided into Instruction and Data caches. $*$ by "processor," I mean a CPU "core."

Memory

Register memory:

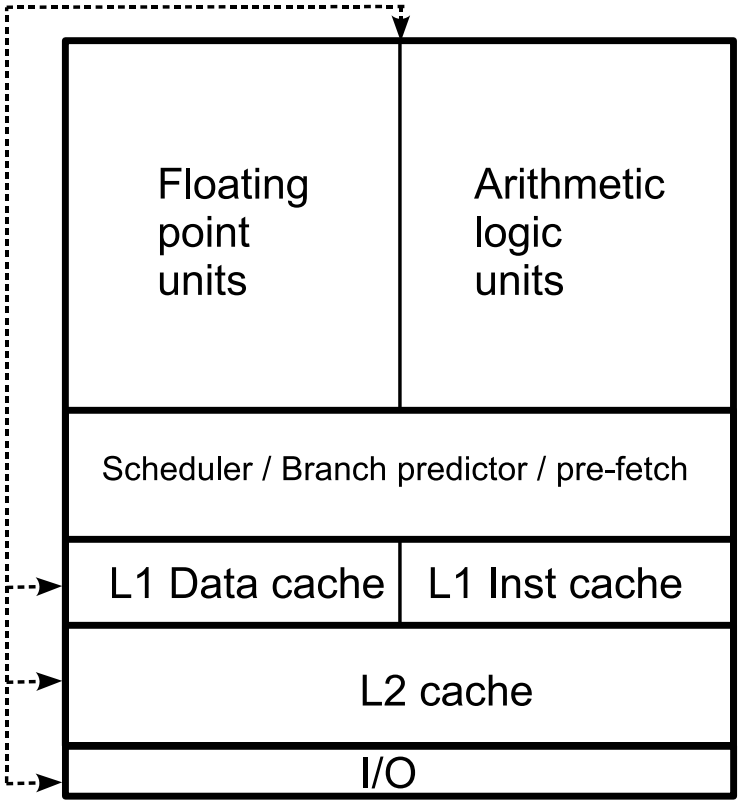
- Smallest ($\lesssim$ 1kB)
- lowest latency (typically 0-3 clock cycles)
- integrated into the execution units

Note: In C/C++, use the `register` keyword to request that loop counters and similar quantities to be stored in CPU registers.

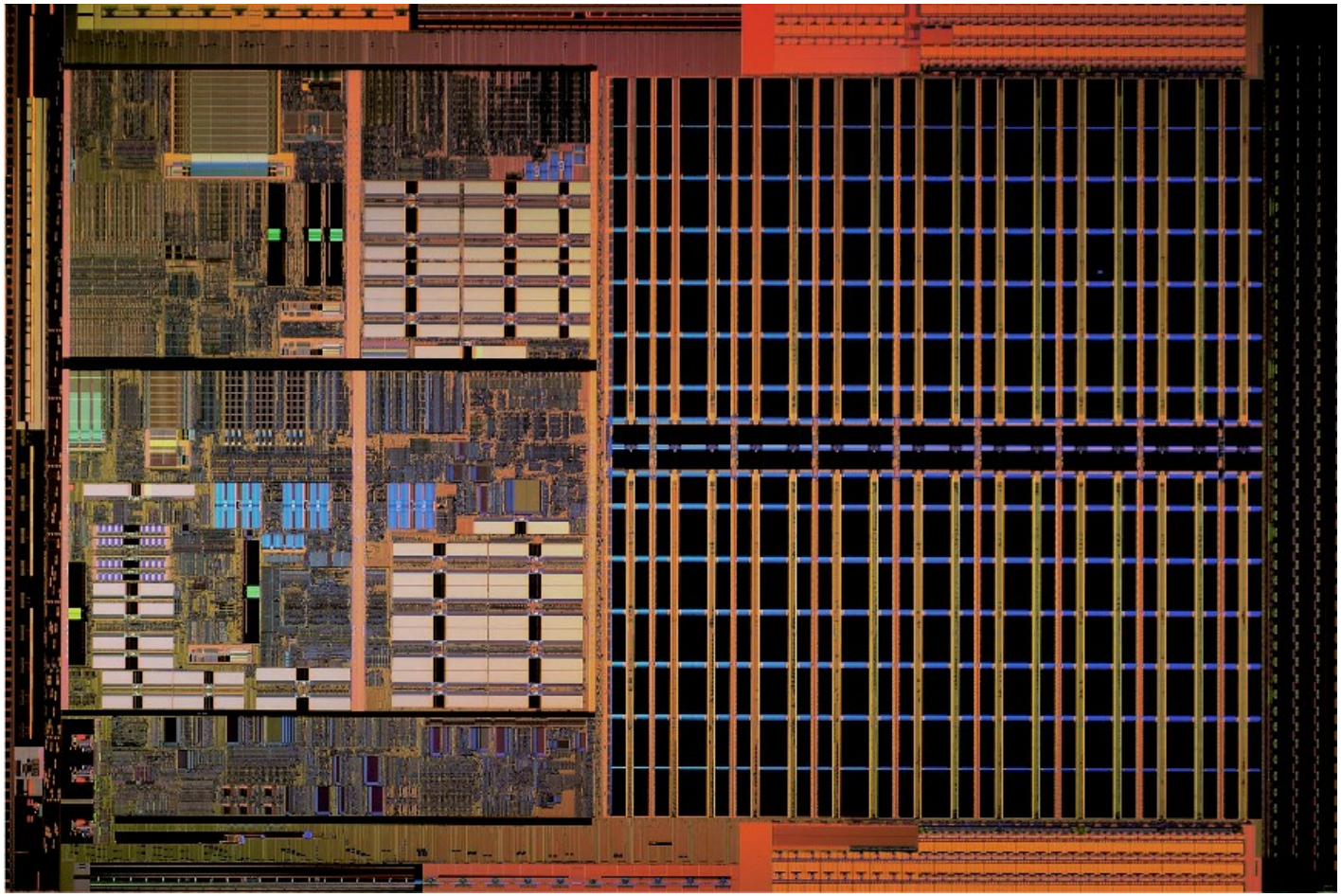
```
register int variable_name
```

Central Processing Units

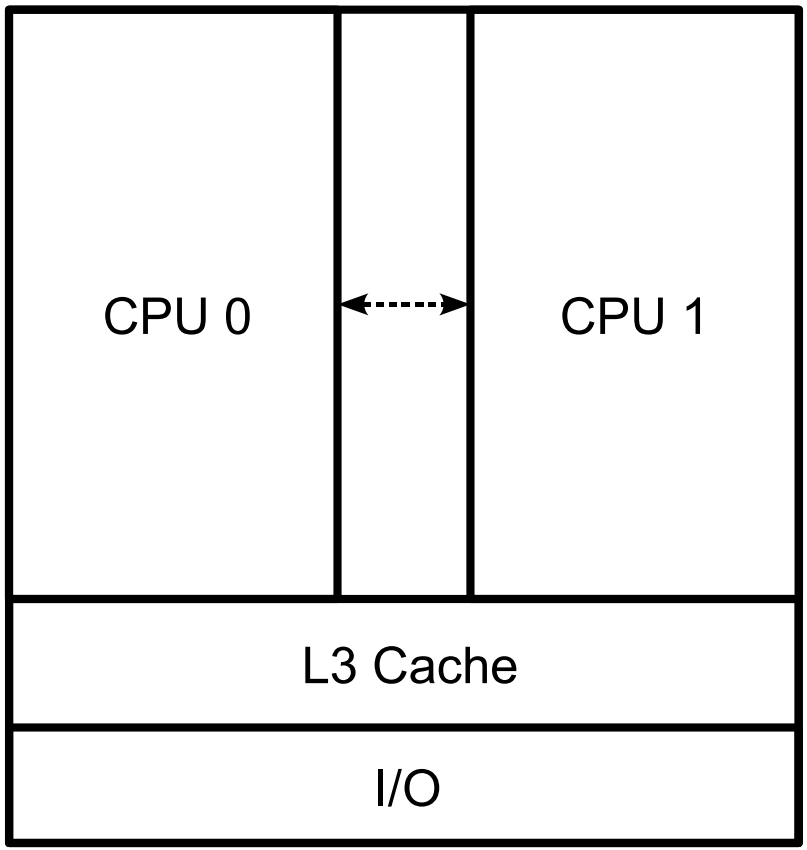
Processor: performs integer and floating point arithmetic and Boolean logic



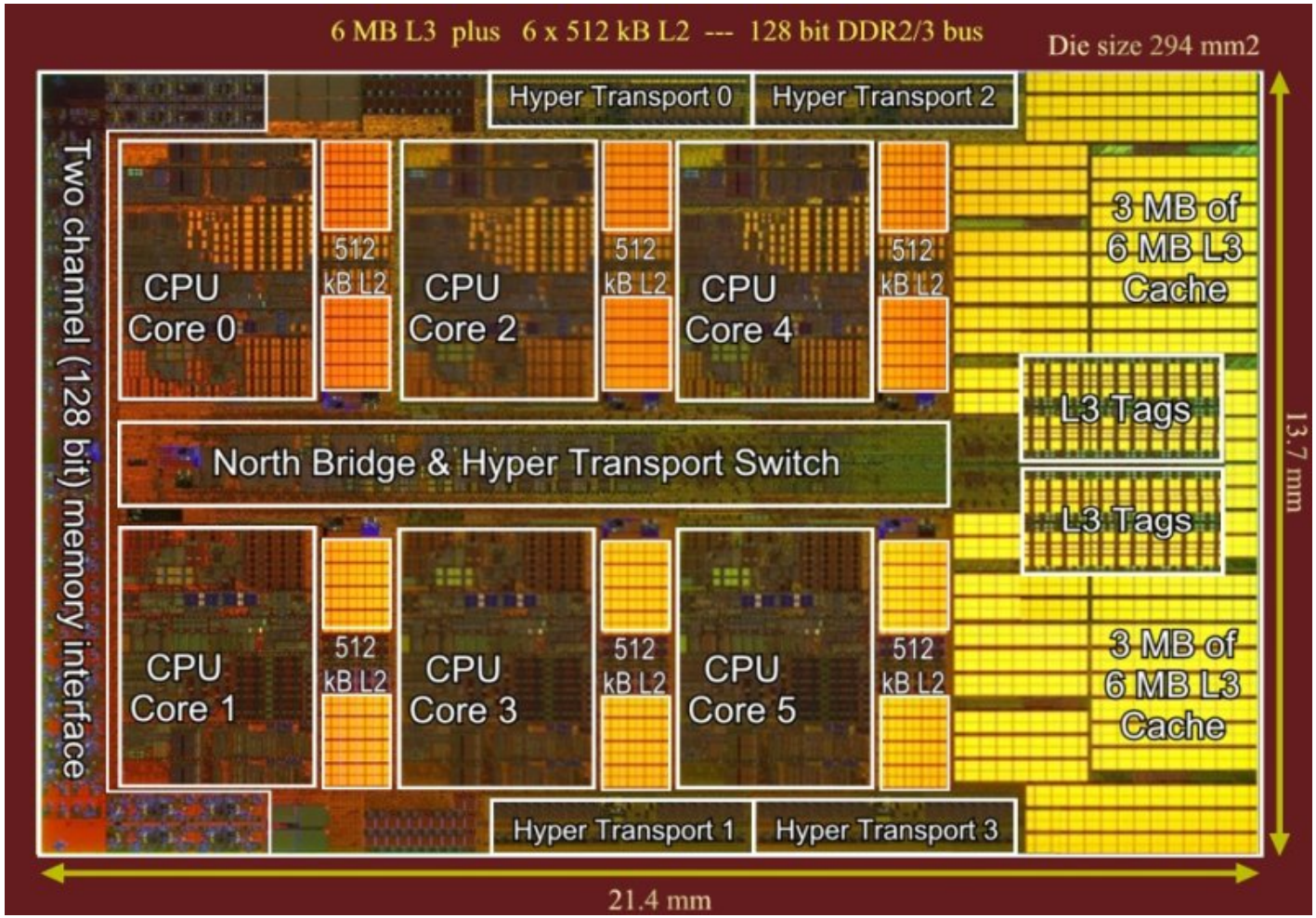
Opteron (AMD k8 "Sledgehammer" microarchitecture)



Multicore CPU



Phenom II Hexacore (k10 "Thuban" microarchitecture)



Vector vs. Scalar

Two extremes: scalar vs. vector processing

Scalar processors

- Input: 1 instruction and 1 or 2 scalar numbers.
- Output: 1 scalar number.
- This is done at most once each clock cycle.

Example

$$3 + 1 = 4$$

Vector processors

Vector processors

- Input: 1 instruction and n or $2n$ scalar numbers.
- Output: n scalars
- This is executed in 1 clock cycle.
- Several clock cycles are required to set up the calculation.

Example 1

$$\begin{bmatrix} 1 \\ 5 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} 3 \\ 2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

Vector processors

Example 2

$$\begin{bmatrix} 1 \\ 5 \\ \vdots \\ x_n \end{bmatrix} \times \begin{bmatrix} 3 \\ 2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 3 \\ 10 \\ \vdots \\ x_n y_n \end{bmatrix}$$

- No advantage over scalar processors for many tasks.
- Large performance advantage if application is vectorizable and n is sufficiently large.
- Depending on chip design, $n = 64,\ 128,\ 256,\ 512,\ 1024,\ 2048 \ldots 64,000.$

*The majority of modern general-purpose processors are neither scalar nor vector!

Hybrid Schemes

Two concepts: Superscalar & SIMD

Superscalar processors:

- Multiple functional units (ALUs, AGUs, FPUs), coupled with
- Advanced control hardware allows
  - Out of order (OOO) execution
  - Speculative execution
  - Symmetric Multi-threading

Superscalar processors typically also include:

- Execution pipelining
- Branch prediction
- Pre-fetching

Superscalar processors (cont'd)

As a result,

- Superscalar processors can execute multiple instructions per clock cycle.
- Efficiently perform a wide range of tasks well (much faster than a scalar CPU).
- Most general-purpose processors are superscalar (All x86 chips since mid 1990's)

Still not as fast as vector processors for specific tasks...

SIMD

Single Instruction, Multiple Data (SIMD)

- Similar to vector processors, but vector length is very short
- Typically $n = 4$
- Faster than superscalar processor on vectorizable code (unless $n_{SIMD} \leq n_{SS}$)
- More complicated to implement code: requires extra syntax and extra thought (although optimizing compilers can automatically generate code).

Common SIMD Instruction sets (for AMD and Intel processors):

- MMX
- 3D Now!, Extended 3DNow!
- SSE, SSE2, SSE3, SSE4, SSSE3, AVX, and soon AVX2

Further hybridization

Modern mainstream PC and server processors are hybrids

- Superscalar
- Contain Integer and FP SIMD units
- Multicore (more on this later)

What's next?

- SIMD units are beginning to resemble vector processors (larger n)
- Number of cores per CPU increases > 16
- Superscalar processors are being combined with stream processors. (more on that later)

Code Vectorization

Vectorization works well for problems which involve performing the same operations on many pieces of data.

Examples:

- multiplying an array by a scalar: $\mathbf{F} = m\mathbf{a}$
- adding two arrays of scalars: $\mathbf{F} = \mathbf{F}_1 + \mathbf{F}_2$
- adding the constant 5 to an array of scalars: $\mathbf{B} = \mathbf{A} + 5\mathbf{I}$

Pseudocode:

Scalar

```
for(i=0;i<n;i++){
z.v[i]=w.v[i]*u.v[i]+r.v[i];
}
```

Vector

```
z.v = w.v * u.v + r.v;
```

Code Vectorization

Details:

For Matlab / Octave

- Vectorized code runs faster than using `for` loops even though these languages are interpreted rather than compiled.
- Addition, subtraction, and scalar multiplication syntax is intuitive.
- For other operations, use the dot along with an operator rather than writing `for` loops.
  - element-wise multiplication: `u.*v`
  - element-wise division: `u./v`
  - element-wise exponentiation: `u.^3`
- For other operations, see the Matlab vectorization guide.

Code Vectorization

For C/C++

- Vector instructions are implemented via SIMD instruction sets.
- See compiler documentation for syntax.
  - For the popular GNU Compiler Collection (GCC), refer to `http://gcc.gnu.org/onlinedocs/`
- Refer to a SIMD programming reference from AMD or Intel website.
- Good optimizing compilers automatically generate SIMD instructions in trivial situations. For GCC, use

`-O2` or `-O3` and `-march=native`

Parallelization

- Multiple processors share the processing load.
- Break problem into discrete, (temporarily) independent pieces.
- Rather than performing the same instructions simultaneously on a single processor as in SIMD/Vector processing, perform *different* instructions on *different* processors (MIMD).

Examples

- Encode / decode multiple files simultaneously.
- Edit frames of a video or large numbers of images simultaneously.
- Calculate gravitational forces for multiple particles / regions simultaneously.
- Advance multiple particles / regions forward in time simultaneously.

Shared Memory

Each processor has access to the entire memory address space.

Advantages

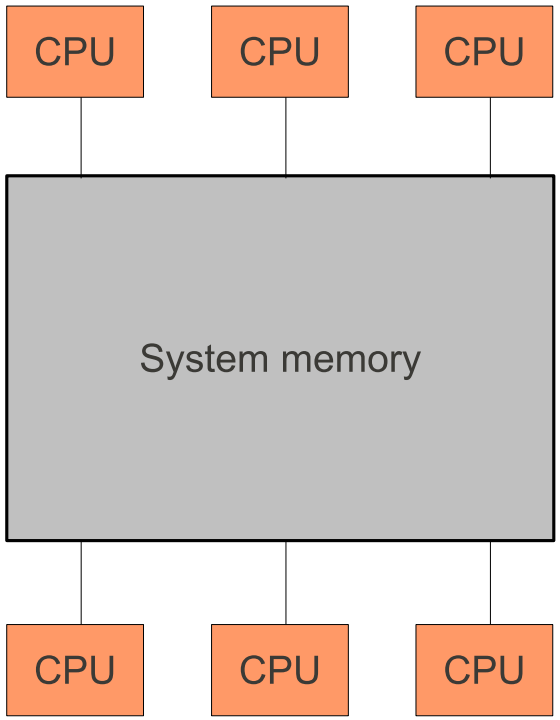
- Easy to program
- High memory bandwidth
- Low memory latency

Disadvantages

- Large systems are very expensive.
- Must be careful to avoid race condition errors.

Code Implementation

- OpenMP, POSIX (p)threads



Shared Memory: SMP vs. NUMA

SMP: Symmetric Multiprocessing

- All processor cores have equal access to shared memory.
- Typically the processor cores share a memory controller / bus.
- Multi-core, single chip processor packages (like the ones in most laptops, desktops, and smart phones).

Shared Memory: SMP vs. NUMA

SMP: Symmetric Multiprocessing

- All processor cores have equal access to shared memory.
- Typically the processor cores share a memory controller / bus.
- Multi-core, single chip processor packages (like the ones in most laptops, desktops, and smart phones).

NUMA: Non-Uniform Memory Access

- Individual cores are assigned only a portion of the shared memory.
- Cores must request to read/write from/to memory addresses that "belong" to other cores.
- Processors either share a PCB or have a very fast, low-latency interconnect.
- Multi-chip, multi-socket systems like servers, workstations, supercomputer nodes.

Shared Memory: OpenMP

OpenMP code example:

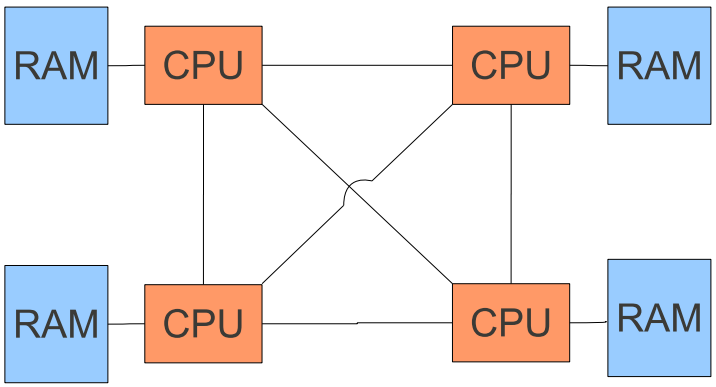
```
unsigned int part;
float vx,vy,vz,v2, K=0;


#pragma omp parallel for schedule(dynamic) default(shared) \
    private(vx,vy,vz,v2,part) reduction(+:K)
for (part=0; part<N; part++)
{
    vx=v[3*part];          //store velocity of "part"
    vy=v[3*part+1];
    vz=v[3*part+2];
    v2=vx*vx+vy*vy+vz*vz; //v^2
    v2*=0.5*mass[part];
    K+=v2;                 //the kinetic energy
}
```

Distributed (private) Memory

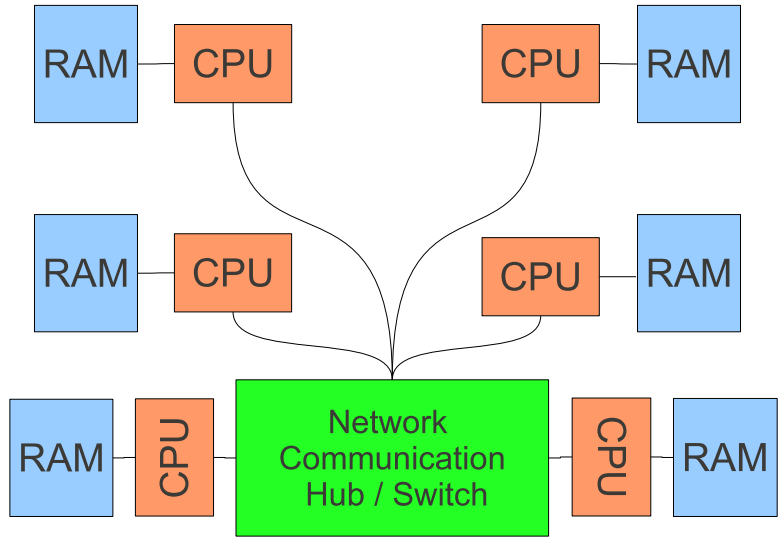
Each processor has its own private memory address space

- Processors must communicate in order to share data.
- There are many different communication schemes.

Point-to-point:



Switching Network:



Distributed Memory

Advantages

- Systems are relatively inexpensive.
- Systems can easily scale to very large sizes.
- Race condition / memory conflicts are not an issue.

Disadvantages

- High memory latency / low bandwidth when communicating between processors.
- Requires more thought about memory management / communication.
- Makes parallelization more difficult in some cases.
- Software development progresses slowly.

Code Implementation

- MPI (Message Passing Interface)

MPI Pseudo Example

```
MPI::Init ( argc, argv ); // start MPI. The following code will run on all threads:

p = MPI::COMM_WORLD.Get_size (); // Get the number of processes.
id = MPI::COMM_WORLD.Get_rank (); // Get the individual process ID.

// send from 0 to all

if ( id == 0 ) MPI::COMM_WORLD.Send ( &message, count, date_type, destination, tag );

// receive from 0
if ( id > 0 ) MPI::COMM_WORLD.Recv ( &message, count, data_type, source, tag, status );


MPI::Finalize (); // stop MPI
```

Stream Processors

Idea:

- Combine many SIMD processors in parallel on a single chip
- Use minimal control hardware (rely on programmer and compiler)

Stream processors are massively parallel. Some contain hundreds of SIMD units.

- Cell processor
- General purpose graphics processing units (GPGPUs)
  - nVidia: Geforce, Quadro, Tesla
  - AMD: Radeon, FireGL, FirePro
- Languages
  - nVidia: CUDA, OpenCL
  - AMD: OpenCL

A New Standard: OpenACC

As of November 2011, a new API is available for using GPGPUs: **OpenACC**

Rather than writing low-level CUDA or OpenCL code, the compiler will automatically generate GPGPU code based on compiler directives very much like OpenMP:

```
#pragma acc directive-name [clause [[,] clause]...]
```

Not yet supported by any compilers. Support is planned for PGI, Cray, and CAPS compilers

More Hybrids

In practice...

- High-performance codes use vectorization and parallelization simultaneously
  - Stream processors take this idea to the extreme
- A mixture of shared and distributed memory systems is used in most supercomputers:
  - Each compute node is a (NUMA+SMP) shared memory system
    - Example: A multi-socket motherboard with multiple multi-core processor packages and a GPGPU
  - Compute nodes are networked together using a high-speed interconnect
    - Interconnect examples: Gigabit Ethernet, Infiniband, Myrinet

Optimization

Basic concept:

- The performance of a computer is limited by its slowest components.
- The performance of an algorithm is limited by its slowest step.

Optimization

Basic concept:

- The performance of a computer is limited by its slowest components.
- The performance of an algorithm is limited by its slowest step.

Thus:

- Avoid using the slowest hardware components when possible.
  - when slow components cannot be avoided, use them efficiently.

Optimization

Basic concept:

- The performance of a computer is limited by its slowest components.
- The performance of an algorithm is limited by its slowest step.

Thus:

- Avoid using the slowest hardware components when possible.
  - when slow components cannot be avoided, use them efficiently.
- Identify slow / inefficient pieces of code using benchmarking.
- Focus effort on optimizing the **slowest** operations.

Hardware Bottlenecks

Main hardware bottlenecks:

- System memory
  - high latency
  - low bandwidth
- Inter-processor & inter-node communication
  - high latency
  - low bandwidth
  - significant problem with distributed memory systems

When you must use these, do so wisely!

Efficient use of Memory

- Store data compactly (bit-fields, bit arrays, & unions can help)
- Do not store unnecessary results
- Use `register` keyword where appropriate

Efficient use of Memory

- Store data compactly (bit-fields, bit arrays, & unions can help)
- Do not store unnecessary results
- Use `register` keyword where appropriate

Principles to remember:

- sequential data is often accessed faster than non-sequential data (remember, the pre-fetch is not psychic)
- localized data is often accessed faster than fragmented data

Efficient use of Memory

- Store data compactly (bit-fields, bit arrays, & unions can help)
- Do not store unnecessary results
- Use `register` keyword where appropriate

Principles to remember:

- sequential data is often accessed faster than non-sequential data (remember, the pre-fetch is not psychic)
- localized data is often accessed faster than fragmented data

Implications:

- sorting algorithms (e.g. quicksort) can sometimes improve execution efficiency
- physical objects which are spatially close should be kept close in memory.

Efficient use of Memory

Be aware of the difference between Row-major order vs. column-major order when using arrays:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Using Row-major order, this is stored in linear memory as

1	2	3	4	5	6

Using Column-major order, the same 2-D array is stored as

1	4	2	5	3	6

Efficient use of Memory

Row-major order:

- C / C++
- Python
- Pascal
- Java
- Ada
- Modula-2
- most high-level languages

Column-major order:

- FORTRAN
- CUDA
- IDL
- MATLAB / GNU Octave

Efficient use of Memory

Passing arguments to functions / subroutines:

- Avoid creating functions with large numbers of arguments.
  - Combine arguments into an array or structure if possible; pass a pointer to the array / struct.

Efficient use of Memory

Passing arguments to functions / subroutines:

- Avoid creating functions with large numbers of arguments.
  - Combine arguments into an array or structure if possible; pass a pointer to the array / struct.
- Avoid passing large arguments (like entire objects or structures) by *value*.
  - passing by *reference* eliminates the need to make a copy.

Efficient Communication

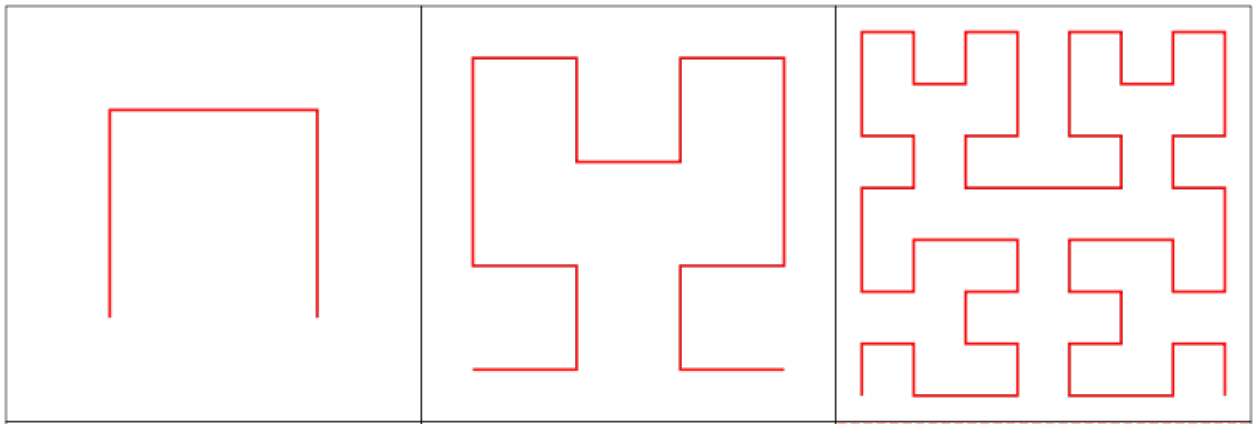
Keep communication to a minimum

- separate problem into sections which require minimal data from other sections.

Example from N-body Methods: Domain decomposition

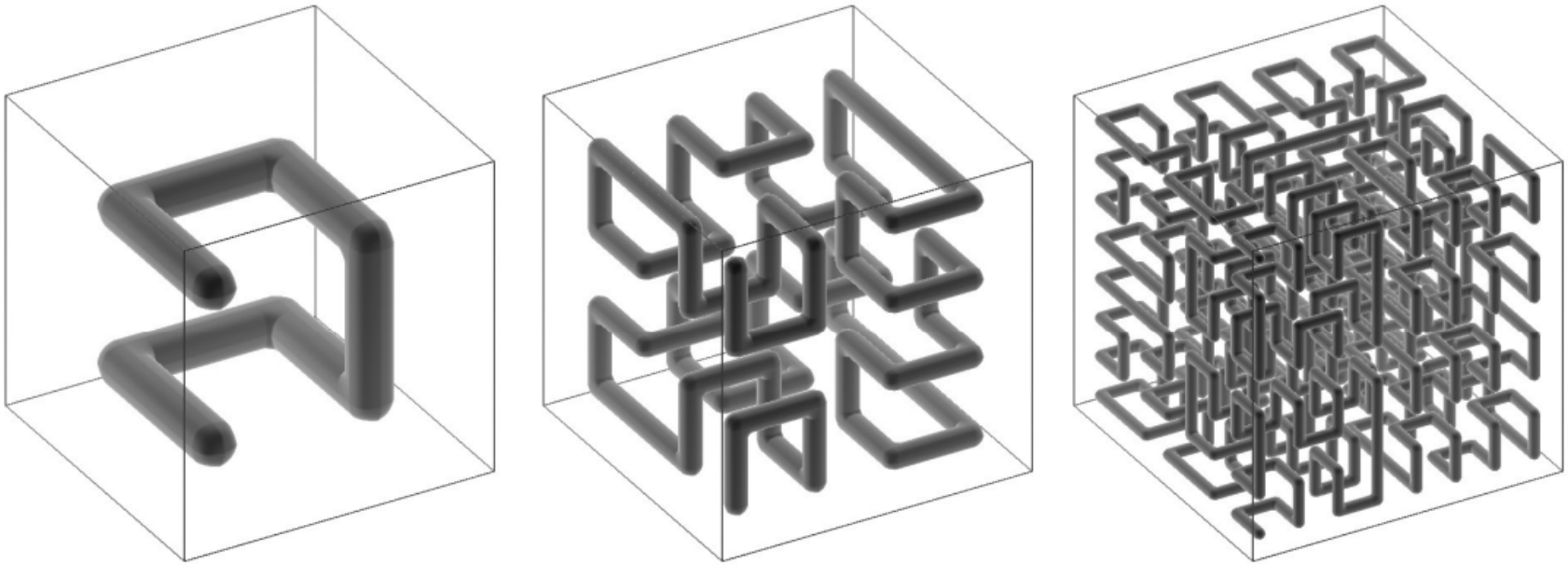
- Use space-filling curve to divide problem among processors

Hilbert Curve:



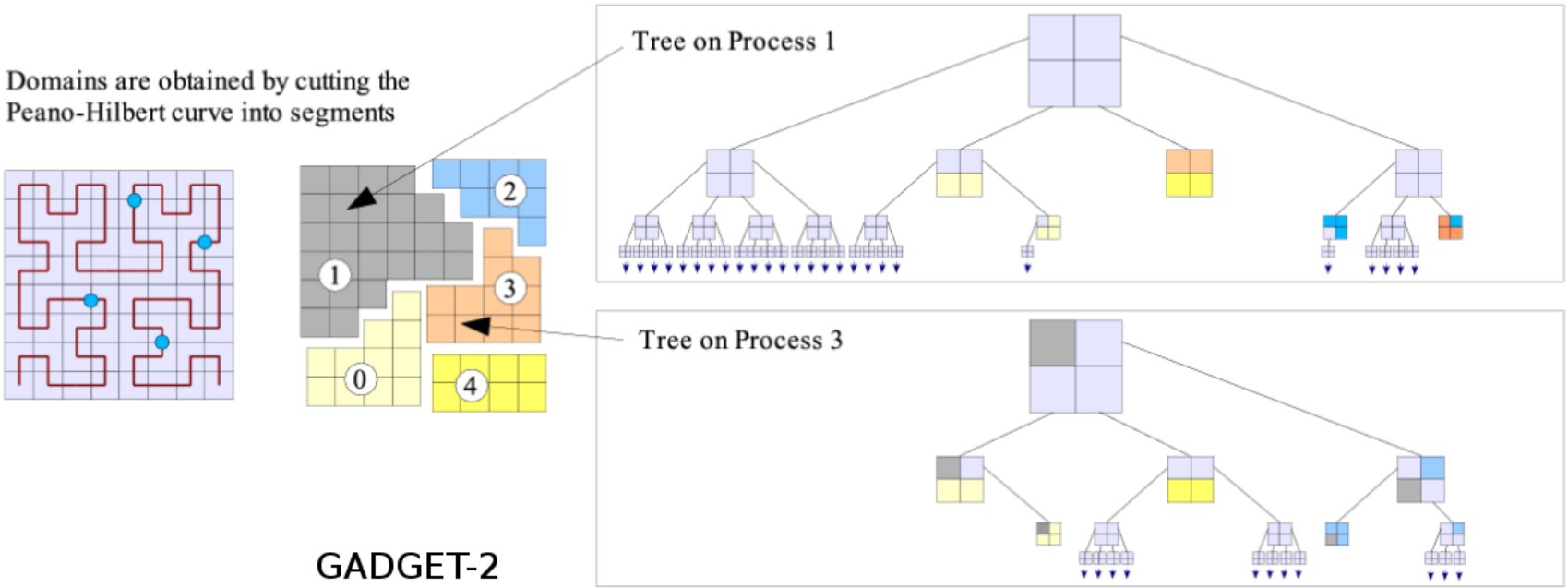
Efficient Communication

Peano-Hilbert Curve:



Efficient Communication

Peano-Hilbert Curve:



This sort of domain decomposition has multiple benefits:

- Regions which are close in space are handled by individual processors
- Particles close in space are close in memory
- Requires message-passing only for long-range force calculations
- Achieves good load balance
  - processors share work-load more evenly
  - no overly-busy or idle processors

Software Bottlenecks

Use a **profiler** to identify the slowest part of your code...

- Profilers analyze program performance in detail.

Software Bottlenecks

Use a **profiler** to identify the slowest part of your code...

- Profilers analyze program performance in detail.
- In the case of GNU (GCC), the main profiler is **GProf**.
  1. compile and link using the `-pg` flag.
  2. run the compiled executable.
  3. run `gprof executable_file > output_file`
  4. Examine the output.

Software Bottlenecks

Use a **profiler** to identify the slowest part of your code...

- Profilers analyze program performance in detail.
- In the case of GNU (GCC), the main profiler is **GProf**.
  1. compile and link using the `-pg` flag.
  2. run the compiled executable.
  3. run `gprof executable_file > output_file`
  4. Examine the output.
- The **Valgrind** suite is also quite useful. Valgrind contains:
  - Memcheck – detects memory-management problems
  - Cachegrind – a cache profiler
  - Callgrind – adds call graph creation to Cachegrind
  - Massif – a heap profiler

Efficiency Tips

There is overhead involved in creating a function call: **Minimize the number of function calls**—especially within `while` and `for` loops.

Efficiency Tips

There is overhead involved in creating a function call: **Minimize the number of function calls**—especially within `while` and `for` loops.

- Reorganize the algorithm.
- Simplify the logic (simplify mathematical expressions algebraically).
- Compute once and save the result for later use (if possible).
- Write short functions inline or use the **inline** keyword.
  - these allow for manual optimization and more aggressive compiler optimization.

Efficiency Tips

Approximation: sometimes an approximation is good enough.

- Using look-up tables + interpolation is sometimes faster than repeated computation of expensive functions.
- Alternatively, a low-order expansion in terms of orthogonal polynomials (e.g., Taylor series or Chebyshev expansion) might be accurate enough in some cases.

Efficiency Tips

Approximation: sometimes an approximation is good enough.

- Using look-up tables + interpolation is sometimes faster than repeated computation of expensive functions.
- Alternatively, a low-order expansion in terms of orthogonal polynomials (e.g., Taylor series or Chebyshev expansion) might be accurate enough in some cases.

Example: $\sin(x)$

For $|x| \leq 1.0$, it is typically sufficient to compute

`x - 0.166666667*x*x*x.`

This is usually faster than computing `sin(x)`

Note: Chebyshev expansion tends to be more accurate than Taylor when >2 terms are used.

Efficiency Tips

Small integer exponents:

- Computing g^n as g*g*g*g*g*$\cdots$*g for $n < 10 - 20$ tends to be faster than using `pow(g,n)`.
- Also try setting g2 = g*g. Then

$$g^n = \text{g2*g2*}\cdots\text{*g2(or *g, if n is odd)}$$

- or create a cascade of such expressions.

Efficiency Tips

Division vs. Multiplication:

- Multiplication tends to be faster than division.
- `0.5*g` generally computes faster than `g/2`.
- If you need to divide many numbers by the variable `k`, compute

```
ki = 1.0/k
```

- then replace instances of `(expression)/k` with `(expression)*ki`.

Efficiency Tips

Small integer multiples:

- On some hardware, addition is significantly faster than multiplication.
  - computing $3g$ as g+g+g might be faster than computing 3*g.
- test your target machine's performance to see which form is faster.

Efficiency Tips

Comparison operations: $>$,$<$,$<=$, $>=$

- Comparisons of floating point numbers tend to be quite slow.
  - **avoid when possible** (or find a trick to speed up the comparison).

Efficiency Tips

Comparison operations: $>$,$<$,$<=$, $>=$

- Comparisons of floating point numbers tend to be quite slow.
  - **avoid when possible** (or find a trick to speed up the comparison).
- Comparisons of unsigned integers tend to be faster than comparisons of signed integers.
  - **use** `unsigned int` **for counters and indices**.*
- Equality and negated equality ($==$, $!=$) are often faster than $<$,$>$,$<=$, $>=$.

*Some OpenMP implementations require loop counters to be `int`.

Efficiency Tips

Appropriate data types:

- Computations involving larger data types use more memory bandwidth, more cache space, and are performed more slowly than low-precision data type computations.
  - Only use double precision when it is really needed.
  - Use long integers only when absolutely needed.
  - Use `short int` in place of `int` when possible.

Efficiency Tips

Appropriate data types:

- Computations involving larger data types use more memory bandwidth, more cache space, and are performed more slowly than low-precision data type computations.
  - Only use double precision when it is really needed.
  - Use long integers only when absolutely needed.
  - Use `short int` in place of `int` when possible.
- In general: Use the smallest data type that will suffice.

Efficiency Tips

Perform tests / experiments:

- When optimizing, test multiple versions of code. Results may be counter-intuitive
  - Reducing the number of total calculations can speed up a loop...
  - BUT: reducing the number of calculations while increasing memory usage may slow the algorithm.
  - ALWAYS run experiments to see which code version is faster.

Efficiency Tips

Perform tests / experiments:

- When optimizing, test multiple versions of code. Results may be counter-intuitive
  - Reducing the number of total calculations can speed up a loop...
  - BUT: reducing the number of calculations while increasing memory usage may slow the algorithm.
  - ALWAYS run experiments to see which code version is faster.
- Test different compiler optimization flags (and different compilers: GCC, LLVM*, Open64, Intel, Oracle, PGI, Cray, CAPS, PathScale, IBM).

* LLVM does not yet support OpenMP.

Extreme Efficiency Madness

The highest efficiency is obtained by carefully hand-coding and targeting code for a specific microarchitecture...

Learn to code in assembly language!

Programming From the Ground Up
by Jonathan Bartlett

Read optimization guides & white papers written by CPU manufacturers.

Productivity: Debuggers

With a debugger, you can perform **many** debugging tasks quickly.
Most commonly:

Productivity: Debuggers

With a debugger, you can perform **many** debugging tasks quickly. Most commonly:

- Identify specific locations in the code (breakpoints) for the execution to pause.
- Identify conditions (watches) that will trigger the execution to pause.

Productivity: Debuggers

With a debugger, you can perform **many** debugging tasks quickly. Most commonly:

- Identify specific locations in the code (breakpoints) for the execution to pause.
- Identify conditions (watches) that will trigger the execution to pause.

When the code is paused, you can:

- examine the values of all variables.
- manually change the values of variables.
- manually call functions (essentially insert code) to change the values of variables.
- step through the code one line at a time.

Productivity: Debuggers

In the GNU world, the main debugger is `gdb`.

- compile using the `-g` and `-O0` flags.

Productivity: Debuggers

In the GNU world, the main debugger is `gdb`.

- compile using the `-g` and `-O0` flags.
- Load the program using: `gdb ./executable_name`

Productivity: Debuggers

In the GNU world, the main debugger is `gdb`.

- compile using the `-g` and `-O0` flags.
- Load the program using: `gdb ./executable_name`
- At `(gdb)` prompt, set a breakpoint using: `break line#`

Productivity: Debuggers

In the GNU world, the main debugger is `gdb`.

- compile using the `-g` and `-O0` flags.
- Load the program using: `gdb ./executable_name`
- At `(gdb)` prompt, set a breakpoint using: `break line#`
- Set a watch condition using: `watch condition`

Productivity: Debuggers

In the GNU world, the main debugger is `gdb`.

- compile using the `-g` and `-O0` flags.
- Load the program using: `gdb ./executable_name`
- At `(gdb)` prompt, set a breakpoint using: `break line#`
- Set a watch condition using: `watch condition`
- To run the program in gdb: `run`

Productivity: Debuggers

In the GNU world, the main debugger is `gdb`.

- compile using the `-g` and `-O0` flags.
- Load the program using: `gdb ./executable_name`
- At `(gdb)` prompt, set a breakpoint using: `break line#`
- Set a watch condition using: `watch condition`
- To run the program in gdb: `run`
- To examine a variable: `print variable_name`

Productivity: Debuggers

In the GNU world, the main debugger is `gdb`.

- compile using the `-g` and `-O0` flags.
- Load the program using: `gdb ./executable_name`
- At `(gdb)` prompt, set a breakpoint using: `break line#`
- Set a watch condition using: `watch condition`
- To run the program in gdb: `run`
- To examine a variable: `print variable_name`
- To change a variable: `call variable_name = expression`

Productivity: Debuggers

In the GNU world, the main debugger is `gdb`.

- compile using the `-g` and `-O0` flags.
- Load the program using: `gdb ./executable_name`
- At `(gdb)` prompt, set a breakpoint using: `break line#`
- Set a watch condition using: `watch condition`
- To run the program in gdb: `run`
- To examine a variable: `print variable_name`
- To change a variable: `call variable_name = expression`
- To call a function: `call function(arg1, arg2,...)`

Productivity: Debuggers

- To continue to the next breakpoint/watch: `continue`

Productivity: Debuggers

- To continue to the next breakpoint/watch: `continue`
- Execute the next line of code: `next` (or `n`)

Productivity: Debuggers

- To continue to the next breakpoint/watch: `continue`
- Execute the next line of code: `next` (or `n`)
- Execute the next line and follow function calls: `step` (or `s`)

Productivity: Debuggers

- To continue to the next breakpoint/watch: `continue`
- Execute the next line of code: `next` (or `n`)
- Execute the next line and follow function calls: `step` (or `s`)
- To execute a shell command: `shell command`

Productivity: Debuggers

- To continue to the next breakpoint/watch: `continue`
- Execute the next line of code: `next` (or `n`)
- Execute the next line and follow function calls: `step` (or `s`)
- To execute a shell command: `shell command`
- For help on a gdb command: `help command`

Productivity: Debuggers

- To continue to the next breakpoint/watch: `continue`
- Execute the next line of code: `next` (or `n`)
- Execute the next line and follow function calls: `step` (or `s`)
- To execute a shell command: `shell command`
- For help on a gdb command: `help command`
- For general help: `help`

Productivity: Debuggers

- To continue to the next breakpoint/watch: `continue`
- Execute the next line of code: `next` (or `n`)
- Execute the next line and follow function calls: `step` (or `s`)
- To execute a shell command: `shell command`
- For help on a gdb command: `help command`
- For general help: `help`
- To exit: `quit`

Productivity: Automatic Documentation

Documentation Generators...

- Examine your source code and automatically document its structure.

Productivity: Automatic Documentation

Documentation Generators...

- Examine your source code and automatically document its structure.
- Use special comments and directives in the source code to add high-level documentations for files, functions, classes, variables.

Productivity: Automatic Documentation

Documentation Generators...

- Examine your source code and automatically document its structure.
- Use special comments and directives in the source code to add high-level documentations for files, functions, classes, variables.
- Create HTML pages, images, PDF documents, & man pages describing the code details and program usage.

Productivity: Automatic Documentation

Documentation Generators...

- Examine your source code and automatically document its structure.
- Use special comments and directives in the source code to add high-level documentations for files, functions, classes, variables.
- Create HTML pages, images, PDF documents, & man pages describing the code details and program usage.

Popular doc gens: Doxygen, ROBODoc, Sphynx, Epydoc

Productivity: Revision Control

Revision Control Systems... (version control, revision management)

- Track source modifications.

Productivity: Revision Control

Revision Control Systems... (version control, revision management)

- Track source modifications.
- Facilitate collaboration between multiple authors.

Productivity: Revision Control

Revision Control Systems... (version control, revision management)

- Track source modifications.
- Facilitate collaboration between multiple authors.
- Stores entire history of each source file.
  - The user can "check out" any version of the source.
  - Bugs / performance regressions can be tracked more easily.

Productivity: Revision Control

Revision Control Systems... (version control, revision management)

- Track source modifications.
- Facilitate collaboration between multiple authors.
- Stores entire history of each source file.
  - The user can "check out" any version of the source.
  - Bugs / performance regressions can be tracked more easily.
- Multiple versions (branches) of the source can be worked on simultaneously.

Productivity: Revision Control

Revision Control Systems... (version control, revision management)

- Track source modifications.
- Facilitate collaboration between multiple authors.
- Stores entire history of each source file.
  - The user can "check out" any version of the source.
  - Bugs / performance regressions can be tracked more easily.
- Multiple versions (branches) of the source can be worked on simultaneously.
- Branches can be merged fairly straightforwardly.

Productivity: Revision Control

Revision Control Systems... (version control, revision management)

- Track source modifications.
- Facilitate collaboration between multiple authors.
- Stores entire history of each source file.
  - The user can "check out" any version of the source.
  - Bugs / performance regressions can be tracked more easily.
- Multiple versions (branches) of the source can be worked on simultaneously.
- Branches can be merged fairly straightforwardly.
- Can be used to manage any text-based document format (i.e. LaTeX documents and SVG images)

Productivity: Revision Control

Two Models

- Client-server: the repository is on a server.
  - Subversion

Productivity: Revision Control

Two Models

- Client-server: the repository is on a server.
  - Subversion
- Distributed: everyone has their own copy of the repository.
  - Git, Mercurial, Bazaar

Productivity: Revision Control

Two Models

- Client-server: the repository is on a server.
  - Subversion
- Distributed: everyone has their own copy of the repository.
  - Git, Mercurial, Bazaar

Repository hosting sites:

- Bitbucket, GitHub, Gitorious, Google Code, SourceForge.

Productivity: Revision Control

Two Models

- Client-server: the repository is on a server.
  - Subversion
- Distributed: everyone has their own copy of the repository.
  - Git, Mercurial, Bazaar

Repository hosting sites:

- Bitbucket, GitHub, Gitorious, Google Code, SourceForge.
- It's also easy to host your own (very easy to install on GNU/Linux Apache server).

Productivity: IDEs

Integrated development environments (IDEs)

- More than simply a text editor with syntax highlighting and code formatting.

Productivity: IDEs

Integrated development environments (IDEs)

- More than simply a text editor with syntax highlighting and code formatting.
- IDEs can create a model of your code:
  - Check for errors on-the-fly (non-trivial syntax errors).
  - Auto-completion of syntax.

Productivity: IDEs

Integrated development environments (IDEs)

- More than simply a text editor with syntax highlighting and code formatting.
- IDEs can create a model of your code:
  - Check for errors on-the-fly (non-trivial syntax errors).
  - Auto-completion of syntax.
- Show Doxygen comments in pop-up help. (hover over a function / class name to get info)

Productivity: IDEs

Integrated development environments (IDEs)

- More than simply a text editor with syntax highlighting and code formatting.
- IDEs can create a model of your code:
  - Check for errors on-the-fly (non-trivial syntax errors).
  - Auto-completion of syntax.
- Show Doxygen comments in pop-up help. (hover over a function / class name to get info)
- Assist in code refactorization.

Productivity: IDEs

Integrated development environments (IDEs)

- More than simply a text editor with syntax highlighting and code formatting.
- IDEs can create a model of your code:
  - Check for errors on-the-fly (non-trivial syntax errors).
  - Auto-completion of syntax.
- Show Doxygen comments in pop-up help. (hover over a function / class name to get info)
- Assist in code refactorization.
- Integrate with profilers, debuggers, revision management.

Productivity: IDEs

The "best" IDEs for the GNU/Linux platform:

- Qt Creator
- KDevelop
- NetBeans
- Code::Blocks
- Eclipse
- Anjuta
- IDLE (for Python)
- Kile (similar to an IDE, but exclusively for LaTeX / TeX)

Productivity: Scripting

Using a scripting language can greatly improve productivity

- Python and the Bourne Shell language are very useful (especially Python!)
- Write scripts for common tasks to avoid boring repetitive work.
- Use scripts to create a streamlined workflow.

Note:

- The `iPython` interpreter can make you more productive!
- The `joblib` Python package allows you to easily run many jobs in parallel.

Productivity: Data Backup

Avoid losing data due to hard drive failure, disaster, or theft!
Always make back-up copies.

The best tool for the job is rsync:

- A powerful command line tool for copying (synchronizing) data.
- Can copy file permissions and all file attributes (modification date, etc.)
- Verifies that the copied data is uncorrupted.
- Only transfers differences (transmits deltas).
- Can backup data over an `ssh` tunnel! (create a remote copy)

Productivity: Data Backup

- You can use `crontab + cron` to schedule and run remote backups at a regular interval.
  1. Create `ssh` (public + private) key pair for source and destination computers.
  2. Create a shell script to run `rsync`.
  3. Make a `cron` job to execute the shell script.

A good combination of `rsync` options:

```
rsync -apvztclEoge ssh src/ usrname@dest/
```

Add the "`--delete`" option to make a perfect copy.

Productivity: UNIX Utils

UNIX Utilities: Getting help on UNIX / GNU/Linux without Google...

`help command` – display information about the built-in `command`.
`man utility` – display the utility's online manual.
`apropos keyword` – search all online manuals for "`keyword`".
`whatis keyword` – display online manual short descriptions.
`info [keyword]` – display info documents.
`type command` – show how the shell interprets `command`.
`which [-a] command` – show the location of `command`.

Productivity: UNIX Utils

Some useful commands and utilities to try:

Basic: `cd`, `ls`, `pwd`, `stat`, `cp`, `mv`, `rm`, `mkdir`, `rmdir`, `shred`, `date`, `echo`, `tree`, `touch`, `ln`, `history`
Display / search / modify / filter: `grep`, `locate`, `find`, `watch`, `awk`, `head`, `tail`, `more`, `less`, `cat`, `tac`, `sort`, `rev`, `split`, `csplit`, `cut`, `fold`, `sed`, `comm`, `cmp`, `diff`, `sdiff`
Misc: `top`, `ps`, `pgrep`, `mpstat`, `w`, `df`, `who`, `users`, `lsof`, `tar`, `gzip`, `gunzip`, `wget`, `export`, `env`, `uptime`, `uname`, `ifconfig`, `ping`, `free`, `df`, `du`, `cron`, `crontab`, `reboot`, `halt`, `mount`, `chroot`, `su`, `chown`, `chmod`, `chgrp`

Acknowledgments

E. Gabriella Canalizo For facilitating the lectures.

Robert Weigel (M.S. advisor)

John Wallin for sharing his N-body / HPC notes

Rainald Löhner for teaching me CFD and HPC

Volker Springel & Lars Hernquist for significant contributions to the field

Till Tantau - The author of the Beamer LaTeX class.

Further Reading

Fluid Dynamics G.K. Batchelor

Galactic Dynamics Binney & Tremaine

Computer Simulation Using Particles Hockney & Eastwood

Numerical Recipes Press, Teukolsky, Vetterling, Flannery

Springel notes `http://www.aip.de/summerschool2006/` (Third week)

TreeSPH Lars Hernquist, Neal Katz, 1989, ApJS, Vol. 70, p. 419-446

GADGET-2 Volker Springel, 2005, MNRAS, Vol. 364, Issue 4, p. 1105-1134

Lagrangian Meshes Volker Springel, 2009, MNRAS, Vol. 401, Issue 3, p. 791-851