N-Body I

SciNet Parallel Scientific Computing Course

Aug 31 - Sept 4, 2009

N-Body Everywhere

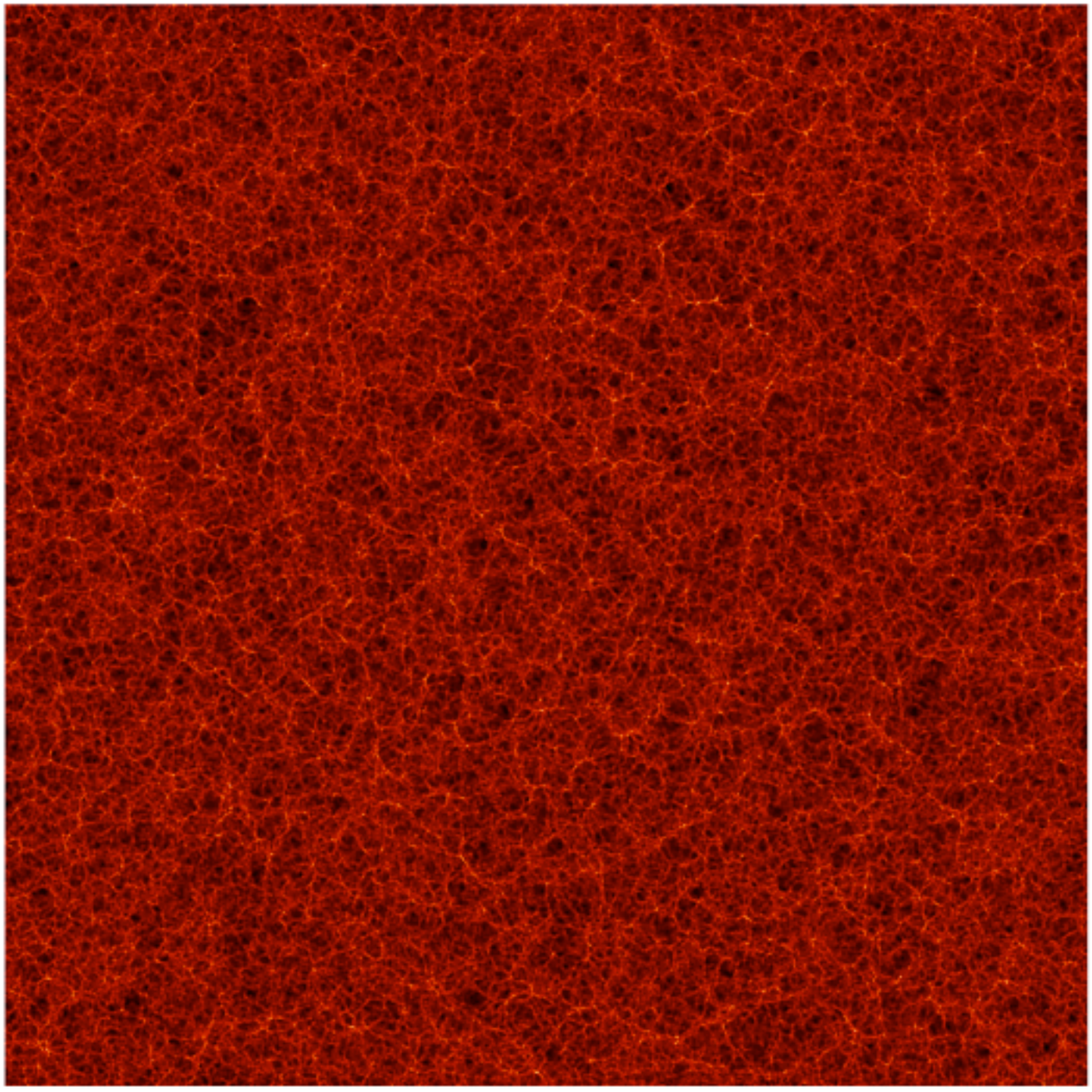
Gravity important in most astrophysical situations.

Cosmology

(At least on large scales)

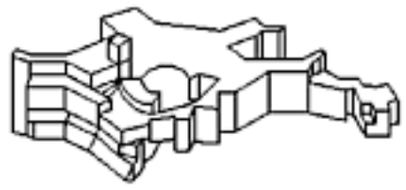
Slice of Hubble-volume simulation. 10^9 particles. Uses Hydra code of Hugh Couchman.

Picture 1% of simulation.



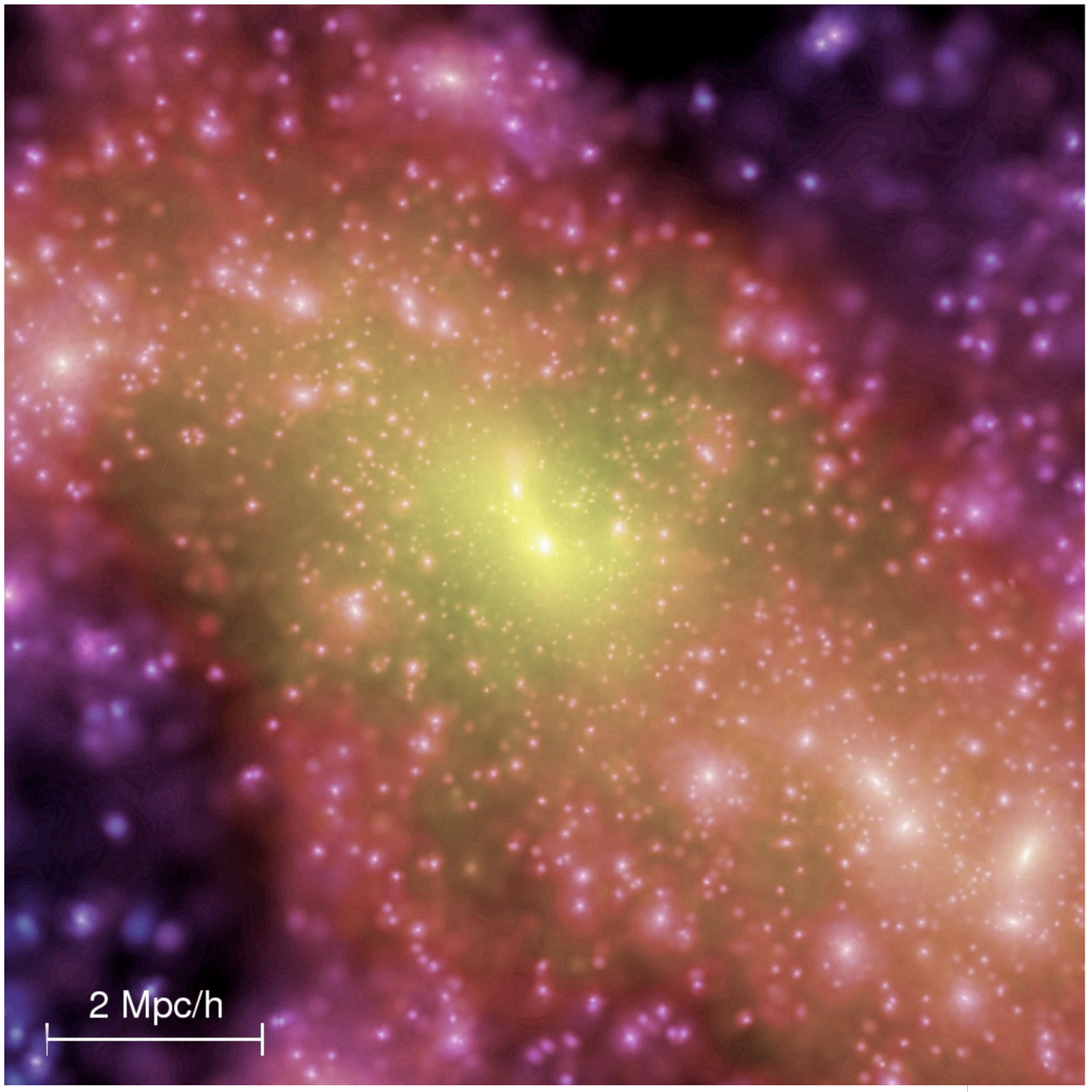
MacFarland, Colberg, White (München), Jenkins, Pearce, Frenk (Durham), Evrard (Michigan), Couchman (London, CA), Thomas (Sussex), Efstathiou (Cambridge), Peacock (Edinburgh)

$2000 \times 2000 \times 20\,(\mathrm{Mpc/h})^3$



Galaxy Clusters

From millennium simulation. 10^{10} particles. Image from Springel *et al.*



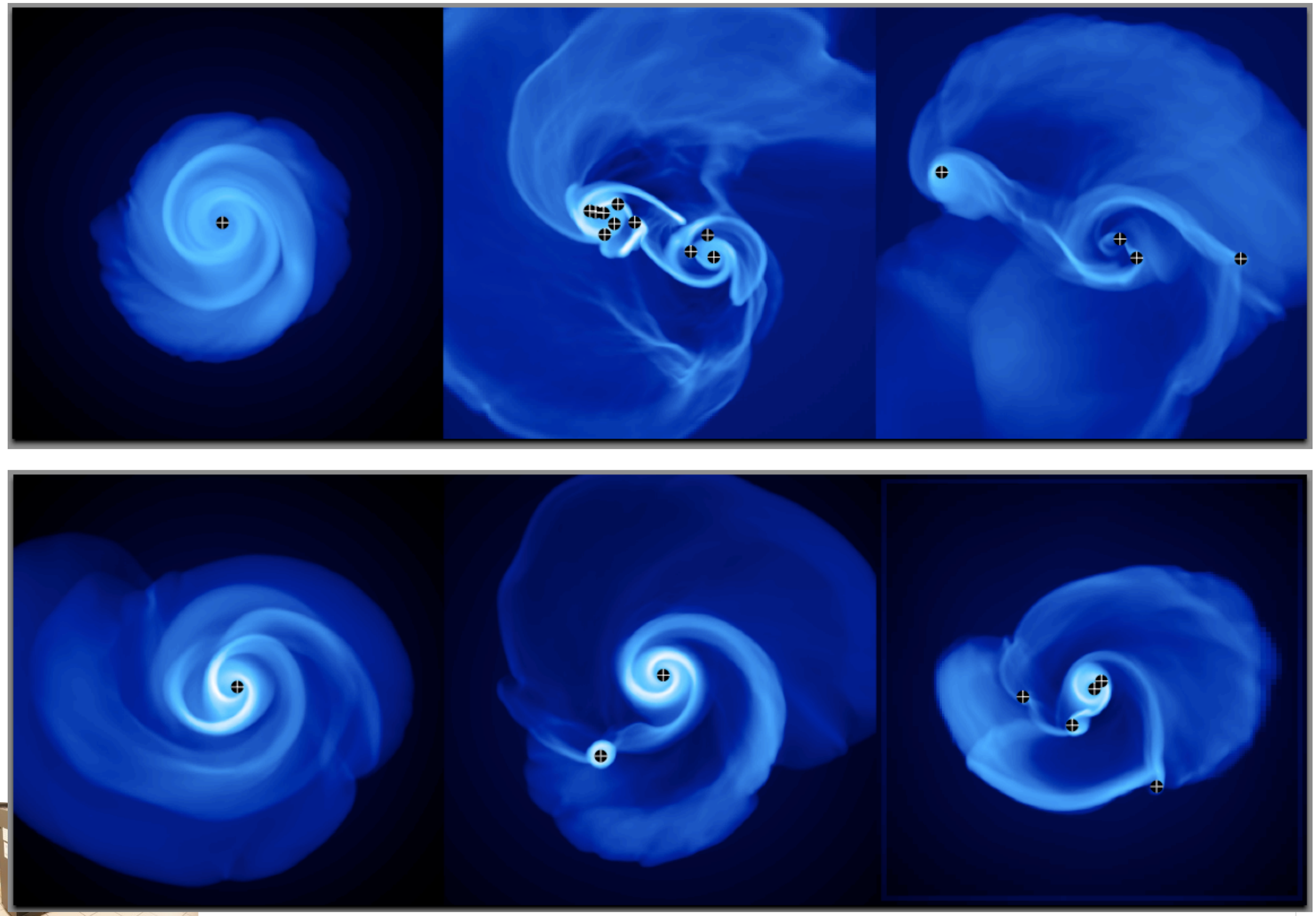
Galaxy Mergers

- Movie from John Dubinski @CITA
- Calculated on CITA McKenzie - predecessor to Sunnyvale.
- (Play move here...)

Globular Clusters

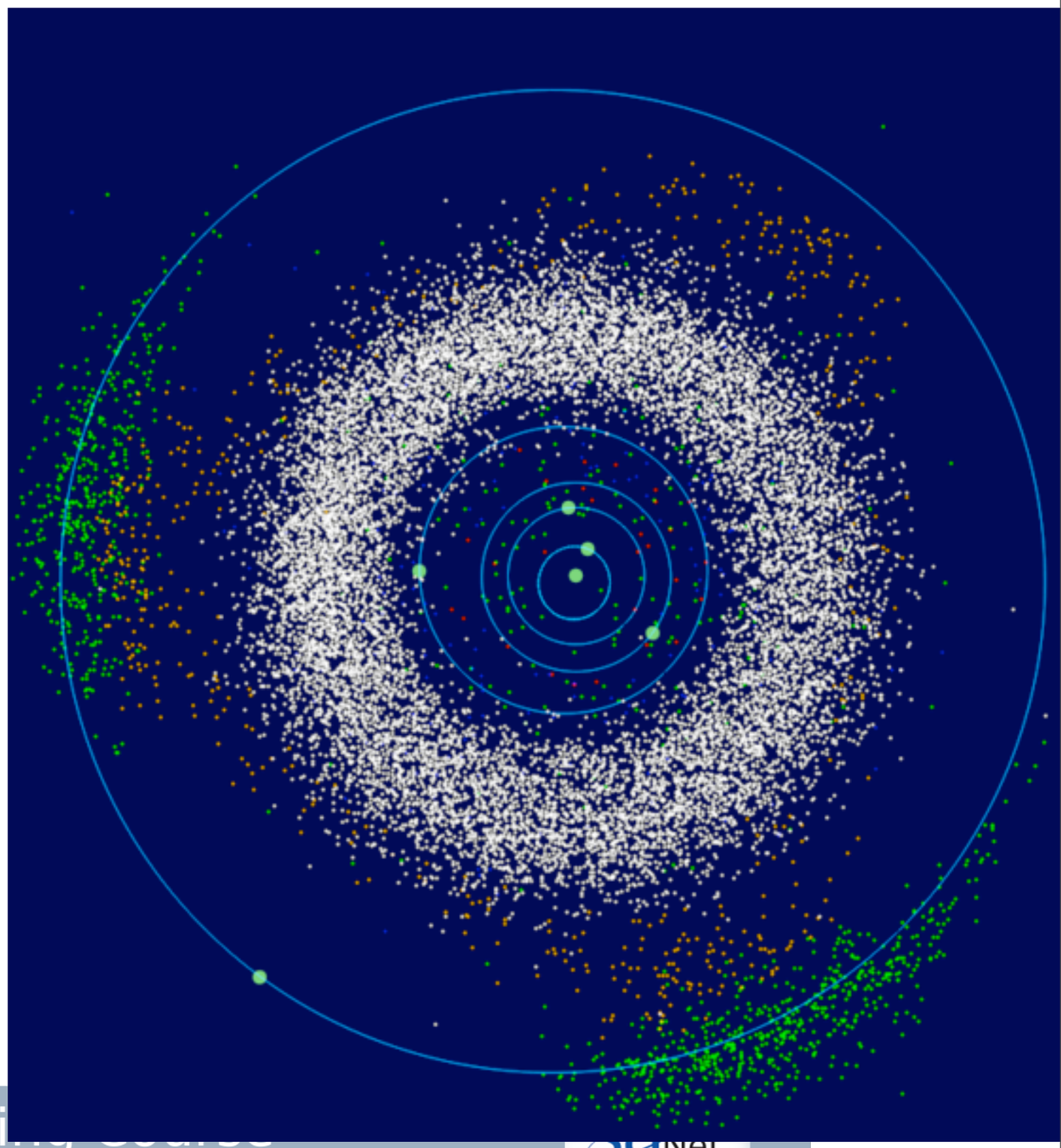
HST Picture of M80

Star Formation



Solar System

- >2 bodies generally not stable.
- Is solar system stable on long times?
- High-precision, conservative n-body sims way to answer.



Gravity Not Easy

- All particles talk to all particles.
- Can use FFTs for large-scale gravity
- See also tree codes, AMR...

Difficulties

- Somehow information about all particles has to make it to all particles. Lots of communication.
- Universe is clumpy - clumps have more particles, higher acceleration.
- Good codes go like *n*log*n* (or even better?)

Nothing Compared to...

Holmberg in 1941 did analog n-body simulation. Took light bulbs - light falls like r^2, just like gravity. Started off 74 light bulbs, used photovoltaic cell to measure light intensity at each light bulb, which is gravity. Calculated motion, physically moved lightbulbs, repeat. Found colliding galaxies merged, took 30 years to verify.

Claimed to get spiral structure. (Experts?)

Direct Summation

- One way to do gravity is direct summation. For every pair of particles, find force between them.
- Have to sum over all pairs of particles: n particles n times means n^2 work.
- Difference between n^2 and $n\log n$ when $n=10^{10}$ is, um, big.
- So, only used for science in special purpose runs, especially solar system dynamics. However, even fancy codes use direct summation, ends up limiting step.
- Need to do 2 things: calculate forces, and update particle positions.

Step 1: Calculate Forces

- Look at nbody.c.
- In general, particles should not be thought of as point masses. Instead, treat as diffuse blobs of matter.
- Force law if $F_{x,i}=\sum(x_i-x_j)/r_{ij}^{3/2}$,$r_{ij}^2=\varepsilon^2+(x_i-x_j)^2$.
- Extra ε^2 softens force between very close particles. So, $F\rightarrow 0$ as $x_i\rightarrow x_j$. Kicks in when $|x_i-x_j| \sim \varepsilon$.

Why Soften?

- Well, let's see. Grab the new nbody. Edit it and set EPS=0.1 at the top.
- Compile and run. Look at the final output column. That is total system energy. How does it behave?
- For ref: columns are iter, step dt, simulation time elapsed, wall-clock time for step, total system energy.
- Now re-do with EPS=0.0. How did the total energy do this time?

What's Going on?

- If there is no softening, particles that interact closely have arbitrarily large accelerations.
- Must track acceleration accurately for accurate solution.
- $\delta v = a\delta t$. If δv in a timestep << $v_{typical}$, system behaves. Max force at $r \sim \varepsilon$, $a \sim Gm/\varepsilon^2$. So, want $\delta t << \varepsilon^2 v_{typical}/Gm$. Can't do this if $\varepsilon = 0$.

Step 2: Update

- We do simplest possible update - at each step, x=x +vδt. v=v+aδt.
- If interpret positions and velocities as staggered by 1/2 timestep, then updating is accurate to 2nd order.

Quick Note on Enery

- Leapfrog technique nominally conserves energy. Energy should be conserved.
- What is energy? A bit tricky if v and r known at different times.
- We ignore time difference. So, will be scatter in reported results when V rapidly transformed to K.
- Energy *does* return to starting value.

Oh, and Step 0: Initial Conditions

- Simulation depends on starting positions.
- Results from bunch of stationary particles look very different from to blobs look different from blobs rotating around each other.
- Put in a few different initial conditions for you.

Command Line Args

- getopt library. Look at code for example.
- Have three classes - spherical collapse, two galaxies at rest, and two orbiting galaxies.
- run nbody -s [1,2,3] for three classes.
- Can also set initial velocity dispersion --vamp, galaxy mass ratio --mass_ratio etc.
- do nbody -h for options.

Let's Watch!

- Cold initial collapse.
- Warm collapse
- Galaxy merger
- Orbiting merger

Now, Let's Talk Nitty-Gritty

- Have to loop through all pairs of particles, summing up pair-wise forces.
- Most expensive bit is $\sqrt{}$ calculation.
- Would like to do as much as possible with $\sqrt{}$ while I have it.
- Single CPU: loop over particles, particles, and dimensions.

Core Code

```
void calculate_forces_fastest(NBody *data, int n)
{
  for (int i=0;i<n;i++) {
    for (int j=0;j<NDIM;j++) {
      data[i].f[j]=0;
    }
    data[i].PE = 0.;
  }

  for (int i=0;i<n;i++){
    for (int j=i+1;j<n;j++) {
      NType rsq=EPS*EPS, dx[NDIM],forcex;
      for (int k=0;k<NDIM;k++){
        dx[k]=data[j].x[k]-data[i].x[k];
        rsq+=dx[k]*dx[k];
      }
      NType ir =1./sqrt(rsq);
      rsq=ir/rsq;
      for (int k=0;k<NDIM;k++)
        {
          forcex=rsq*dx[k] * data[i].mass * data[j].mass * GRAVCONST;
          data[i].f[k] += forcex;
          data[j].f[k] -= forcex;
        }
      data[i].PE -= GRAVCONST * data[i].mass * data[j].mass * ir;
      data[j].PE -= GRAVCONST * data[i].mass * data[j].mass * ir;
    }
  }
}
```

Clear out forces at beginning.

Also going to accumulate potential energy of particles.

Core Code

```
void calculate_forces_fastest(NBody *data, int n)
{
  for (int i=0;i<n;i++) {
    for (int j=0;j<NDIM;j++) {
      data[i].f[j]=0;
    }
    data[i].PE = 0.;
  }

  for (int i=0;i<n;i++){
    for (int j=i+1;j<n;j++) {
      NType rsq=EPS*EPS, dx[NDIM],forcex;
      for (int k=0;k<NDIM;k++){
        dx[k]=data[j].x[k]-data[i].x[k];
        rsq+=dx[k]*dx[k];
      }
      NType ir =1./sqrt(rsq);
      rsq=ir/rsq;
      for (int k=0;k<NDIM;k++)
        {
          forcex=rsq*dx[k] * data[i].mass * data[j].mass * GRAVCONST;
          data[i].f[k] += forcex;
          data[j].f[k] -= forcex;
        }
      data[i].PE -= GRAVCONST * data[i].mass * data[j].mass * ir;
      data[j].PE -= GRAVCONST * data[i].mass * data[j].mass * ir;
    }
  }
}
```

Loop over all particles

Loop over all particles current particle hasn't met.

Core Code

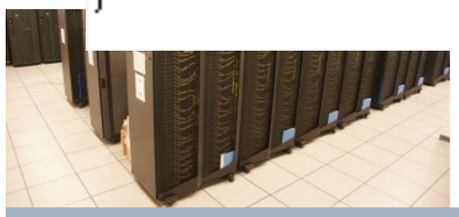
```
void calculate_forces_fastest(NBody *data, int n)
{
  for (int i=0;i<n;i++) {
    for (int j=0;j<NDIM;j++) {
      data[i].f[j]=0;
    }
    data[i].PE = 0.;
  }

  for (int i=0;i<n;i++){
    for (int j=i+1;j<n;j++) {
      NType rsq=EPS*EPS, dx[NDIM],forcex;
      for (int k=0;k<NDIM;k++){
        dx[k]=data[j].x[k]-data[i].x[k];
        rsq+=dx[k]*dx[k];
      }
      NType ir =1./sqrt(rsq);
      rsq=ir/rsq;
      for (int k=0;k<NDIM;k++)
        {
          forcex=rsq*dx[k] * data[i].mass * data[j].mass * GRAVCONST;
          data[i].f[k] += forcex;
          data[j].f[k] -= forcex;
        }
      data[i].PE -= GRAVCONST * data[i].mass * data[j].mass * ir;
      data[j].PE -= GRAVCONST * data[i].mass * data[j].mass * ir;
    }
  }
}
```

Apply force softening

Find errors in all dimensions.

Find r^2



Core Code

```
void calculate_forces_fastest(NBody *data, int n)
{
  for (int i=0;i<n;i++) {
    for (int j=0;j<NDIM;j++) {
      data[i].f[j]=0;
    }
    data[i].PE = 0.;
  }

  for (int i=0;i<n;i++){
    for (int j=i+1;j<n;j++) {
      NType rsq=EPS*EPS, dx[NDIM],forcex;
      for (int k=0;k<NDIM;k++){
        dx[k]=data[j].x[k]-data[i].x[k];
        rsq+=dx[k]*dx[k];
      }
      NType ir =1./sqrt(rsq);
      rsq=ir/rsq;
      for (int k=0;k<NDIM;k++)
        {
          forcex=rsq*dx[k] * data[i].mass * data[j].mass * GRAVCONST;
          data[i].f[k] += forcex;
          data[j].f[k] -= forcex;
        }
      data[i].PE -= GRAVCONST * data[i].mass * data[j].mass * ir;
      data[j].PE -= GRAVCONST * data[i].mass * data[j].mass * ir;
    }
  }
}
```

Find square root, $r^{-3/2}$. Ouch

Loop over dimensions, sum forces on i,j.

Accumulate potential energy.

Homework: Step 1

- Make a copy of nbody.c called nbody_omp.c. OpenMP the force routine without using temporary buffers.
- In its simplest incarnation, to avoid data race, may have to find force of j^{th} particle on i^{th}, but not add force to j^{th}.
- How well does this scale vs. single cpu?

Homework: Step 2

- Make a copy of nbody.c called nbody_buf.c. OpenMP the force routine allowing yourself as much temporary space as you like. Should no longer need to do double work.
- How well are you scaling? If you aren't getting factor of 2, can you think of why? Might OpenMP be able to help you?
- Do a top while running serial and openmp. What is ratio of memory usage?
- What is your estimate of computing to reduction work, and how does it scale with *n*?

Homework: Step 3
(Tricky)

- Ideal code would scale well (no factor of 2) *and* have no large buffers (*i*.e. comparable to total particles per thread). Code also needs to produce correct answers.
- Can you make a code, nbody_nobuf.c that does this?
- Use whatever OpenMP arsenal you like/need: schedules, locks... Concepts from the matrix block-multiply may be useful. You can restrict # of particles to be, say, multiple of 100.
- No particular solution in mind. Be creative!