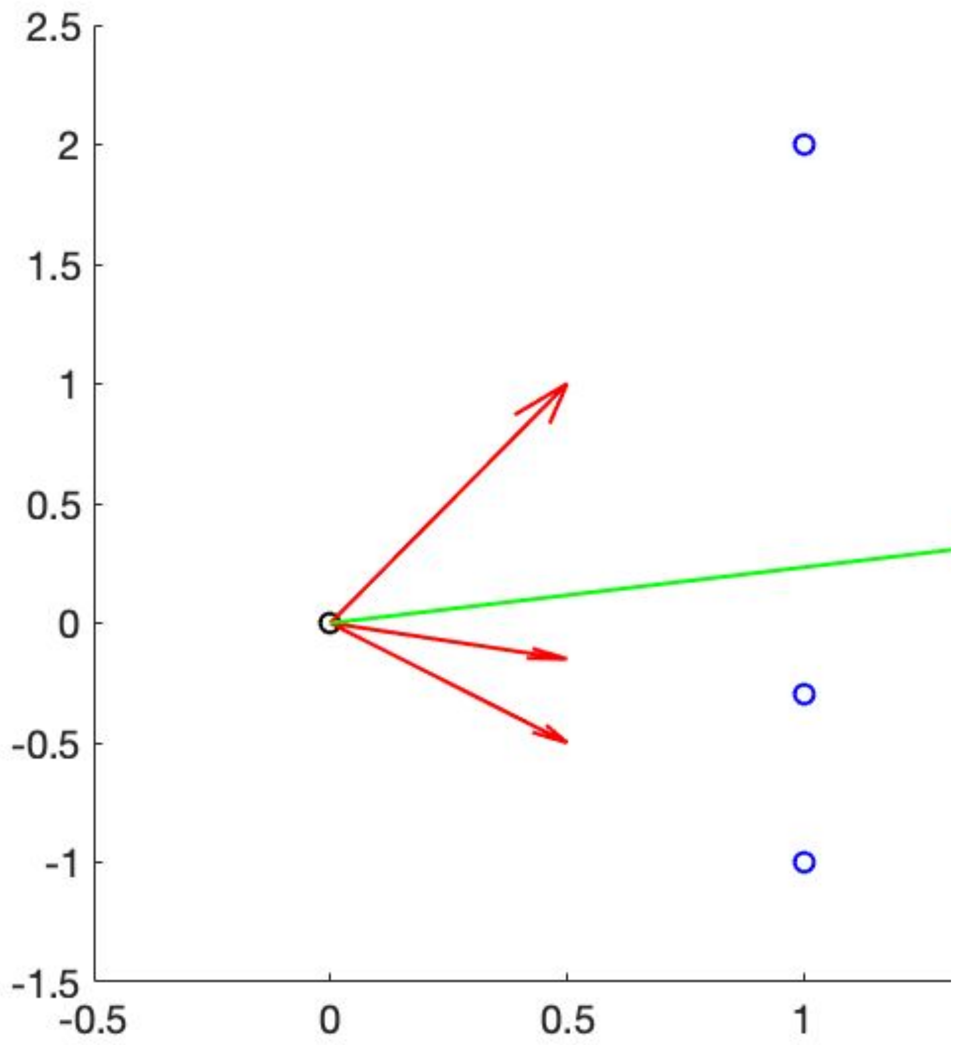
# A simple N-Body Problem Parallelized with OpenMP

# What is the N-body problem

- In the N-body problem, some number of bodies (N) interact with each other.
- Each Body feels the force of the interaction from every other body
- Thus the total force on a body is given by $F_j = \sum_{i \neq j} F_{i,j}$
- If there are N bodies, $(N^2-N)/2$ forces must normally be calculated (red)
- These are summed resulting in a single net force (green)



# Newtonian Mechanics

-

- We start from Newton's laws, namely F=ma (where F is the force, m the mass, and a the acceleration
- As acceleration is the second derivative of position with respect to time we arrive at the differential equation linking displacement to force:

$$\frac{\partial^2 \overrightarrow{r_i(t)}}{\partial t^2} = \frac{\overrightarrow{F_i(t)}}{m_i}$$

- Now we need to develop an expression for force

# Newtonian Gravity

- Different forces have different mathematical expressions
  - For example, force fields with repulsive and attractive terms that often look like $\pm \frac{1}{|\vec{r}|^n}$ or $\pm\, e^{-\alpha|\vec{r}|^n}$ are commonly used in molecular mechanics problems
- For this problem we will be using Newtonian Gravity: $\overrightarrow{F_{i,j}} = \frac{Gm_i m_j}{\left|\overrightarrow{r_{i,j}}\right|^3}\overrightarrow{r_{i,j}}$ where $\overrightarrow{F_{i,j}}$ is the force exerted on object i by object j, G is a universal constant, $m_i$ the mass of object i, and $\overrightarrow{r_{i,j}}$ is the displacement vector from object I to object j.
- In order to find the total force (needed for the differential equation) we need to sum over all the bodies thus the total force on object i can be written as:

$$\overrightarrow{F_{i,j}} = \sum_{j \neq i} \frac{Gm_i m_j}{\left|\overrightarrow{r_{i,j}}\right|^3}\overrightarrow{r_{i,j}}$$

# Deriving Verlet's Method (1/2)

- 
- The displacement of the ith particle can be expanded forward in time as a Taylor series (to 3rd order) as

$$\overrightarrow{r_i(t_0+\Delta t)} = \overrightarrow{r_i(t_0)} + \left[\left(\frac{\partial \overrightarrow{r_i(t)}}{\partial t}\right)_{t=t_0} \cdot \Delta t\right] + \left[\frac{1}{2}\left(\frac{\partial^2 \overrightarrow{r_i(t)}}{\partial t^2}\right)_{t=t_0} \cdot \Delta t^2\right] + \frac{1}{6}\left(\frac{\partial^3 \overrightarrow{r_i(t)}}{\partial t^3}\right)_{t=t_0} \cdot \Delta t^3$$

  Where f is the force, a the acceleration, r the displacement (all vectors), and t is the time and m the mass of the particle

- We can also expand the displacement backward in time:

$$\overrightarrow{r_i(t_0-\Delta t)} = \overrightarrow{r_i(t_0)} - \left[\left(\frac{\partial \overrightarrow{r_i(t)}}{\partial t}\right)_{t=t_0} \cdot \Delta t\right] + \left[\frac{1}{2}\left(\frac{\partial^2 \overrightarrow{r_i(t)}}{\partial t^2}\right)_{t=t_0} \cdot \Delta t^2\right] - \frac{1}{6}\left(\frac{\partial^3 \overrightarrow{r_i(t)}}{\partial t^3}\right)_{t=t_0} \cdot \Delta t^3$$

## Deriving Verlet’s Method (2/2)

- 
- If we add the previous two equations together and solve for $\overrightarrow{r_i(t_0 + \Delta t)}$, we notice that the terms with odd powers of $\Delta t$ cancel and we arrive at:

$$\overrightarrow{r_i(t_0 + \Delta t)} = 2 \cdot \overrightarrow{r_i(t_0)} - \overrightarrow{r_i(t_0 - \Delta t)} + 2\left[\frac{1}{2}\left(\frac{\partial^2 \overrightarrow{r_i(t)}}{\partial t^2}\right)_{t=t_0} \cdot \Delta t^2\right]$$

- Thus the next position depends on the previous two positions and the second derivative at the previous position.
- Recall though that the second derivative is just the acceleration, thus we can simplify this to:

$$\overrightarrow{r_i(t_0 + \Delta t)} = 2 \cdot \overrightarrow{r_i(t_0)} - \overrightarrow{r_i(t_0 - \Delta t)} + 2\sum_{j \neq i} \frac{G m_j}{\left|\overrightarrow{r_{i,j}}\right|^3} \overrightarrow{r_{i,j}}$$

# Writing the vectors using components

- Recall that:
  - $\overrightarrow{r_i(t)} = x(t)\hat{\imath} + y(t)\hat{\jmath} + z(t)\hat{k}$ (here $\hat{\imath}$, $\hat{\jmath}$, and $\hat{k}$ are the unit vectors in Cartesian space) and
  - $\overrightarrow{r_{i,j}(t)} = (x_j(t) - x_i(t))\hat{\imath} + (y_j(t) - y_i(t))\hat{\jmath} + (z_j(t) - z_i(t))\hat{k}$
- Thus we can break the equation we derived for Verlet's method into three equations coupled through the $\overrightarrow{r_{i,j}}$ terms:

$$x_i(t_0 + \Delta t) \ = 2 \cdot x_i(t_0) - x_i(t_0 - \Delta t) + 2 \sum_{j \neq i} \frac{G m_j}{\left|\overrightarrow{r_{i,j}}\right|^3} (x_j(t_0) - x_i(t_0))$$

$$y_i(t_0 + \Delta t) \ = 2 \cdot y_i(t_0) - y_i(t_0 - \Delta t) + 2 \sum_{j \neq i} \frac{G m_j}{\left|\overrightarrow{r_{i,j}}\right|^3} (y_j(t_0) - y_i(t_0))$$

$$z_i(t_0 + \Delta t) \ = 2 \cdot z_i(t_0) - z_i(t_0 - \Delta t) + 2 \sum_{j \neq i} \frac{G m_j}{\left|\overrightarrow{r_{i,j}}\right|^3} (z_j(t_0) - z_i(t_0))$$

# There is a problem however

•

$$x_i(t_0 + \Delta t) \ = 2 \cdot x_i(t_0) - x_i(t_0 - \Delta t) + 2 \sum_{j \neq i} \frac{G m_j}{\left|\overrightarrow{r_{i,j}}\right|^3} (x_j - x_i)$$

- This formula requires we know 2 positions! In general we only know the initial conditions (or 1 position)!
- We can overcome this if we know an initial velocity, the first derivative, $v_i(t_0) = \left(\frac{\partial \overrightarrow{r_i(t)}}{\partial t}\right)_{t=t_0}$ if we use this and we assume the third derivative (the jerk) is 0 we can use our Taylor series to get a second position, for example:

$$x_i(t_0 + \Delta t) \ = x_i(t_0) + v_i(t_0) \cdot \Delta t + \frac{1}{2}\left[\sum_{j \neq i} \frac{G m_j}{\left|\overrightarrow{r_{i,j}}\right|^3} (x_j(t_0) - x_i(t_0))\right] \Delta t^2$$

## The Algorithm

- Load initial conditions
- (Calculate Accelerations) Loop i over the body index form 1 to Nbodies
    - Loop j over the body index from 1 to Nbodies
        - Calculate force between object i and j
        - Add force into the sum of the force on object i
- (Calculate Next Position - Taylor) Loop over the body index from 1 to Nbodies
    - Calculate a second set of x, y, & z coordinates $\Delta t$ in the future (the next time step) using the previous Taylor series expansion
- Loop from time step 2 until the final time index (Ntimes)
    - (Calculate Accelerations) Loop i over the body index form 1 to Nbodies
        - Loop j over the body index from 1 to Nbodies
            - Calculate force between object i and j
            - Add force into the sum of the force on object i
    - (Calculate Next Position - Verlet) Loop over the body index from 1 to Nbodies
            - Calculate the next set of x, y, & z  coordinates using Verlet’s method

If you were to pick one thing to parallelize, with OpenMP, what would it be?

## If you were to pick one thing to parallelize, with OpenMP, what would it be?

Answer: Calculating the Accelerations (two nested loops – it scales as $N^2$)!

# Acceleration Code: Where does the #pragma go?

```
int ibody2
for(int ibody1=0;ibody1<Nbodies;ibody1++) {
        dx=x[index(itime,ibody1,Nbodies)];
        dy=y[index(itime,ibody1,Nbodies)];
        dz=z[index(itime,ibody1,Nbodies)];
        r=sqrt(dx*dx+dy*dy+dz*dz)+1.0e7;
          // 1.0e7 is to soften the potential, prevent "explosions"
        F=(G*10*mass*Nbodies/(r*r*r));
        ax[ibody1]=-F*dx;
        ay[ibody1]=-F*dy;
        az[ibody1]=-F*dz;
        for (ibody2=0;ibody2<ibody1-1;ibody2++) {
                dx=x[index(itime,ibody1,Nbodies)]-
                    x[index(itime,ibody2,Nbodies)];
                dy=y[index(itime,ibody1,Nbodies)]-
                   y[index(itime,ibody2,Nbodies)];
                dz=z[index(itime,ibody1,Nbodies)]-z[index(itime,ibody2,Nbodies)];
                r=sqrt(dx*dx+dy*dy+dz*dz)+1.0e7;
                // 1.0e7 is to soften the potential, prevent "explosions"
                F=(G*mass/(r*r*r));
                ax[ibody1]-=F*dx;
                ay[ibody1]-=F*dy;
                az[ibody1]-=F*dz;}
        // We don't do ibody1=ibody2
        for (ibody2=ibody1+1;ibody2<Nbodies;ibody2++) {
                dx=x[index(itime,ibody1,Nbodies)]-
                       x[index(itime,ibody2,Nbodies)];
                dy=y[index(itime,ibody1,Nbodies)]-
                       y[index(itime,ibody2,Nbodies)];
```

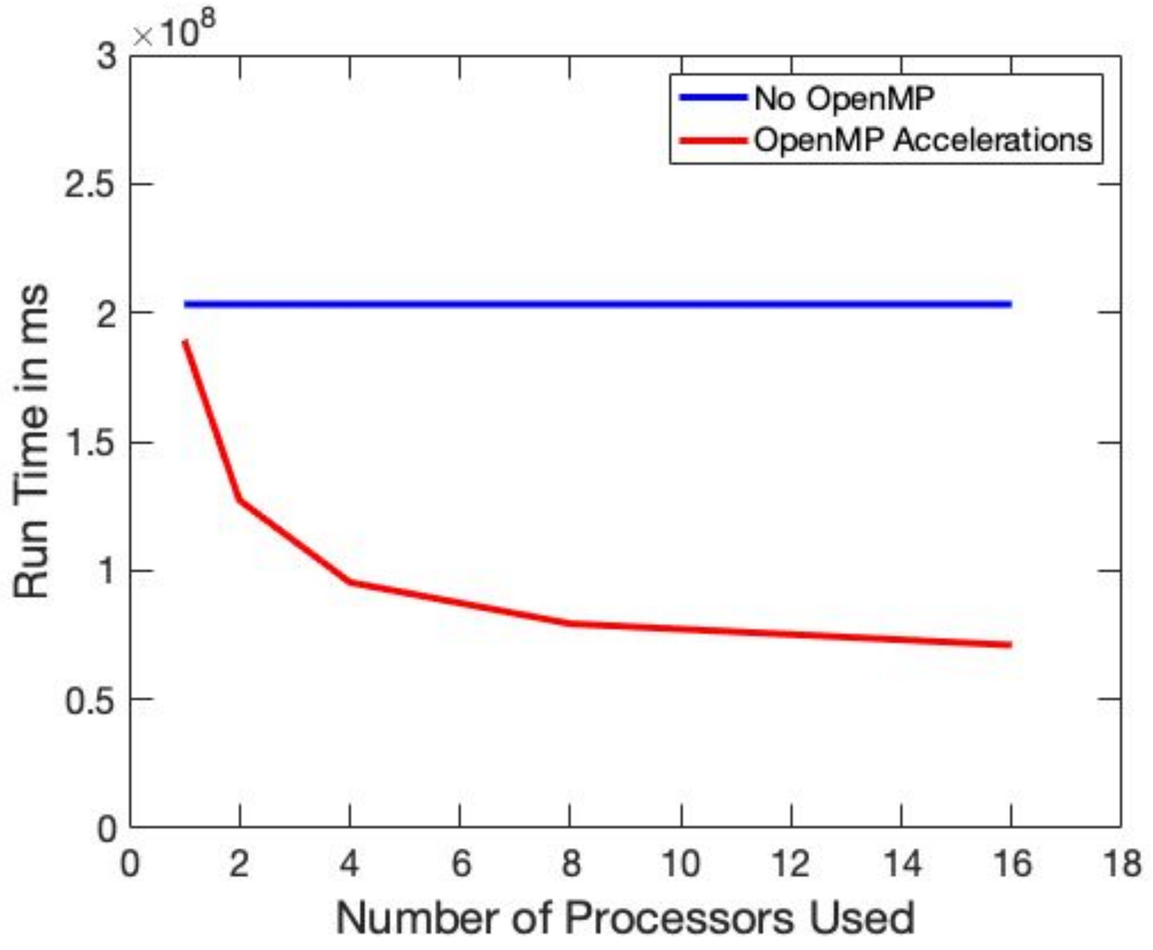
# Acceleration Code: Where does the #pragma go?

#pragma omp parallel for simd private(ibody2,dx,dy,dz,r,F)

```
dx=x[index(itime,ibody1,Nbodies)];
dy=y[index(itime,ibody1,Nbodies)];
dz=z[index(itime,ibody1,Nbodies)];
r=sqrt(dx*dx+dy*dy+dz*dz)+1.0e7;
 // 1.0e7 is to soften the potential, prevent "explosions"
F=(G*10*mass*Nbodies/(r*r*r));
ax[ibody1]=-F*dx;
ay[ibody1]=-F*dy;
az[ibody1]=-F*dz;
for (ibody2=0;ibody2<ibody1-1;ibody2++) {
      dx=x[index(itime,ibody1,Nbodies)]-
          x[index(itime,ibody2,Nbodies)];
      dy=y[index(itime,ibody1,Nbodies)]-
          y[index(itime,ibody2,Nbodies)];
      dz=z[index(itime,ibody1,Nbodies)]-
          z[index(itime,ibody2,Nbodies)];
      r=sqrt(dx*dx+dy*dy+dz*dz)+1.0e7;
      // 1.0e7 is to soften the potential, prevent "explosions"
      F=(G*mass/(r*r*r));
      ax[ibody1]-=F*dx;
      ay[ibody1]-=F*dy;
      az[ibody1]-=F*dz;}
// We don't do ibody1=ibody2
for (ibody2=ibody1+1;ibody2<Nbodies;ibody2++) {
      dx=x[index(itime,ibody1,Nbodies)]-
           x[index(itime,ibody2,Nbodies)];
      dy=y[index(itime,ibody1,Nbodies)]-
```

## Runtime as a function of the number of processors

- We can plot the run time of the code (with the SINGLE pragma versus the number of processors used on BlueWaters
- As you can see the run time decreases nicely with the number of processors



## Can We do Better?

## Position Calculations: Where do the #pragmas go?

```
for (int ibody1=0; ibody1<Nbodies; ibody1++) {
     // vx and vy for cirucular rotation
     vx=-omega*y[index(itime,ibody1,Nbodies)];
     vy=omega*x[index(itime,ibody1,Nbodies)];
     x[index(itimep1,ibody1,Nbodies)]=x[index(itime,ibody1,Nbodies)]+
          vx*dt+0.5*ax[ibody1]*dt*dt;
     y[index(itimep1,ibody1,Nbodies)]=y[index(itime,ibody1,Nbodies)]+
          vy*dt+0.5*ay[ibody1]*dt*dt;
     z[index(itimep1,ibody1,Nbodies)]=z[index(itime,ibody1,Nbodies)]+
          0.5*az[ibody1]*dt*dt;} // vz=0

  // Now the meat!  Verlet integration
  // x(i)=2*x(i-1)-x(i-2)+a(i-1)*dt*dt

  for(itime=1;itime<Ntime-1;itime++){
       acceleration(x,y,z,itime,Nbodies,mass,ax,ay,az);
       itimep1=itime+1;
       itimem1=itime-1;
       for(int ibody1=0;ibody1<Nbodies;ibody1++){
            x[index(itimep1,ibody1,Nbodies)]=
               2*x[index(itime,ibody1,Nbodies)]-
               x[index(itimem1,ibody1,Nbodies)]+ax[ibody1]*dt*dt;
            y[index(itimep1,ibody1,Nbodies)]=
               2*y[index(itime,ibody1,Nbodies)]-
               y[index(itimem1,ibody1,Nbodies)]+ay[ibody1]*dt*dt;
            z[index(itimep1,ibody1,Nbodies)]=
               2*z[index(itime,ibody1,Nbodies)]-
```

# Position Calculations: Where do the #pragmas go?

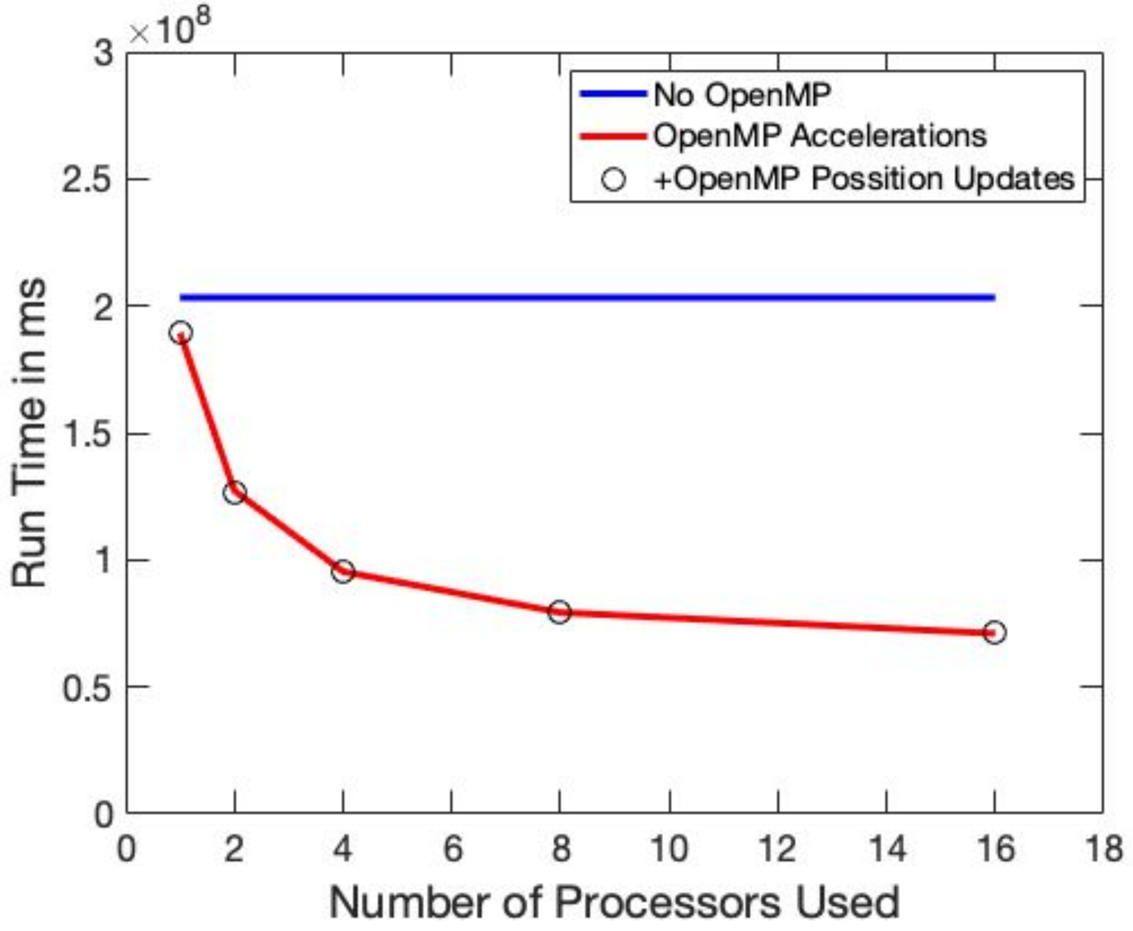
```
#pragma omp parallel for simd private(vx,vy)
for (int ibody1=0; ibody1<Nbodies; ibody1++) {
    // vx and vy for cirucular rotation
    vx=-omega*y[index(itime,ibody1,Nbodies)];
    vy=omega*x[index(itime,ibody1,Nbodies)];
    x[index(itimep1,ibody1,Nbodies)]=x[index(itime,ibody1,Nbodies)]+
        vx*dt+0.5*ax[ibody1]*dt*dt;
    y[index(itimep1,ibody1,Nbodies)]=y[index(itime,ibody1,Nbodies)]+
        vy*dt+0.5*ay[ibody1]*dt*dt;
    z[index(itimep1,ibody1,Nbodies)]=z[index(itime,ibody1,Nbodies)]+
        0.5*az[ibody1]*dt*dt;} // vz=0

  // Now the meat!  Verlet integration
  // x(i)=2*x(i-1)-x(i-2)+a(i-1)*dt*dt

#pragma omp parallel for simd
for(itime=1;itime<Ntime-1;itime++){
    acceleration(x,y,z,itime,Nbodies,mass,ax,ay,az);
    itimep1=itime+1;
    itimem1=itime-1;
    for(int ibody1=0;ibody1<Nbodies;ibody1++){
        x[index(itimep1,ibody1,Nbodies)]=
            2*x[index(itime,ibody1,Nbodies)]-
            x[index(itimem1,ibody1,Nbodies)]+ax[ibody1]*dt*dt;
        y[index(itimep1,ibody1,Nbodies)]=
            2*y[index(itime,ibody1,Nbodies)]-
            y[index(itimem1,ibody1,Nbodies)]+ay[ibody1]*dt*dt;
        z[index(itimep1,ibody1,Nbodies)]=
```

## Runtime as a function of the number of processors

- As before we can plot the run time of the code (both of the SINGLE pragma in the acceleration routine and with the two additional pragmas in the location calculation loops) versus the number of processors used on BlueWaters
- As we can see the run time decreases nicely with the number of processors
- But parallelizing the location calculation loops results in essentially no speedup! 10000 bodies is simply too small to see the effect.



## Is there anything left to Parallelize?

# Initialization Code: Where does the #pragma go?

- The example code uses a Monte Carlo approach to build initial conditions where the mass density decays as $\rho(r)=e^{-ar}$, where r is the density and a is a characteristic scale length

```
for (int ibody1=0; ibody1<Nbodies; ibody1++) {
    prob1=0.0;
    prob2=1.0;
    while (prob2>=prob1) {
        tempx=rand();
          // Recall rand returns an integer between 0 and RAND_MAX
        tempx=((tempx/RAND_MAX)-0.5)*2*LBox;
         // Now tempx holds a number between -LBox and + LBox
        tempy=rand();
        tempy=((tempy/RAND_MAX)-0.5)*2*LBox;
        tempz=rand();
        tempz=((tempz/RAND_MAX)-0.5)*2*LBox;
           // The above code will make our masses scattered
           //evenly throughout the box.
           // We really want the mass to be more concentrated
           // at the center, so we are going to accept
           // or reject with a probability of
           //  exp(-scalelength*r^2/LBox)
           // This code +  the while loop accomplishes this
        tempr=sqrt(tempx*tempx+tempy*tempy+tempz*tempz);
        prob1=exp(-scalelength*tempr/LBox);
        prob2=rand();
```

## It is a trick Question – rand is not thread safe!

- We do need to add a “#pragma omp parallel for” but just adding a parallel for will not result in working code!
- The behavior of rand() is not defined for multiple threads
- Need to instead use rand_r() and a different seed for each thread (we don’t want the threads putting objects at the same coordinates as each other)
- Warning: rand()/rand_r() were used for simplicity here, and because we don’t need good pseudo-random numbers.
- In general rand()/rand_r() are not good enough for scientific codes – but that is beyond the scope of this module

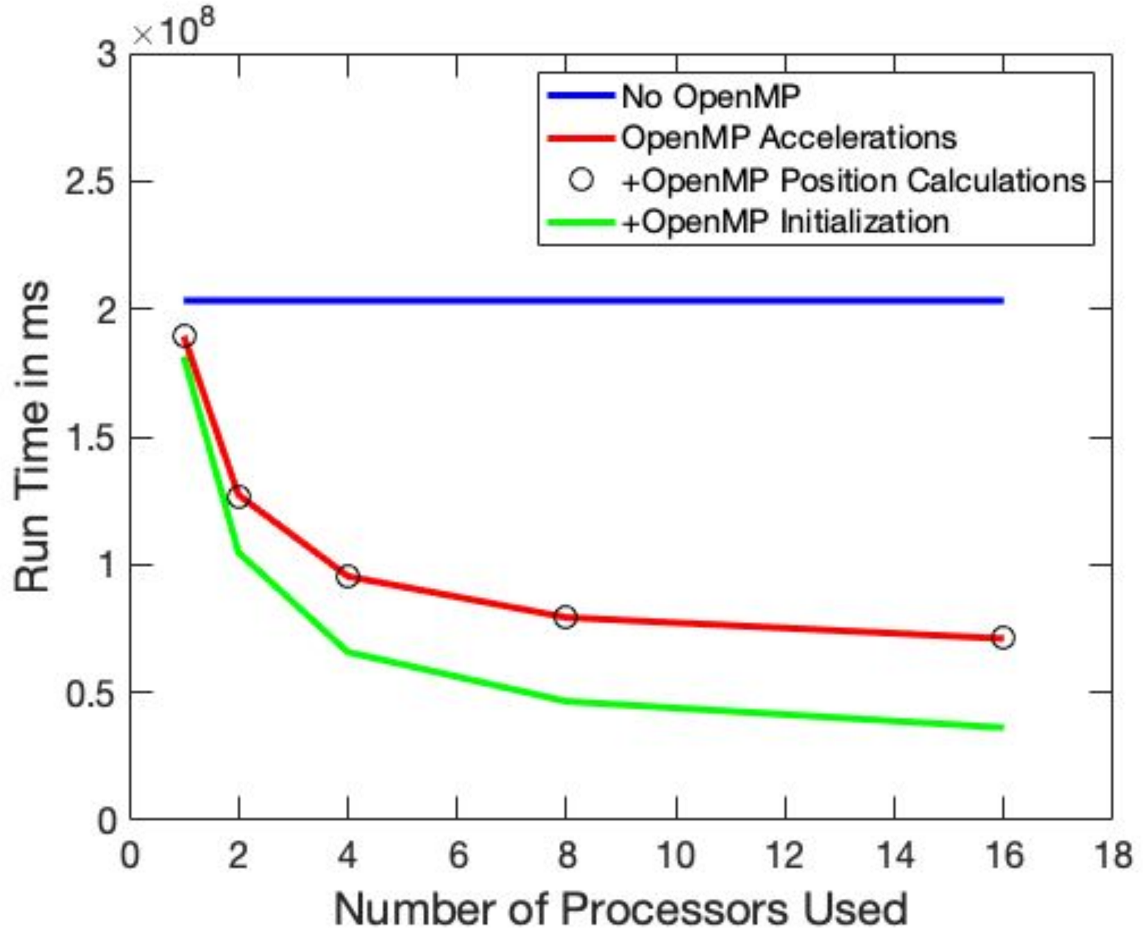
# Initialization Code: The Parallelized version

- Note the “#pragma omp parallel” beginning is separated from the “#pragma omp for”
- This is because each thread must generate its own seed, which this code bases in part on its thread number
- All the calls to rand() have been replaced by calls to rand_r(&seed), the seed must be passed with each call to rand_r()

```
#pragma omp parallel
  private(tempx,tempy,tempz,tempr,prob1,prob2)
  {
  unsigned int seed=t1.tv_usec+omp_get_thread_num();
#pragma omp for
  for (int ibody1=0; ibody1<Nbodies; ibody1++) {
        prob1=0.0;
        prob2=1.0;
        while (prob2>=prob1) {
              tempx=rand_r(&seed);
                // Recall rand returns an integer between 0 and RAND_MAX
              tempx=((tempx/RAND_MAX)-0.5)*2*LBox;
               // Now tempx holds a number between -LBox and + LBox
              tempy=rand_r(&seed);
              tempy=((tempy/RAND_MAX)-0.5)*2*LBox;
              tempz=rand_r(&seed);
              tempz=((tempz/RAND_MAX)-0.5)*2*LBox;
              // The above code will make our masses scattered evenly
              // throughout the box.
              // We really want the mass to be more concentrated at the
              // center, so we are going to accept
              // or reject with a probability of exp(-scalelength*r^2/LBox)
              // This code +  the while loop accomplishes this
              tempr=sqrt(tempx*tempx+tempy*tempy+tempz*tempz);
              prob1=exp(-scalelength*tempr/LBox);
```

## Runtime as a function of the number of processors

- As before we can plot the run time of the code (both of the SINGLE pragma in the acceleration routine and with the two additional pragmas in the location calculation loops) versus the number of processors used on BlueWaters
- As we can see the run time decreases nicely with the number of processors
- But parallelizing the location calculation loops results in essentially no speedup! 10000 bodies is simply too small to see the effect.



## Can we do more?

- Yes, but we’d need to move away from the way we are storing positions and calculating accelerations.
- One of the options is the Barnes-Hut algorithm which stores the positions in an octree which allows one to easily treat collections of distant objects as a single object with their combined mass at their center of mass.
- Another is solving for the gravitational potential (using for instance a spectral method) at each time step and calculating the forces based on that potential.
- Both approaches are beyond the scope of this module
- If you’d like to know more, check out the GalaxySee HPC modules at
  - http://shodor.org/petascale/materials/UPModules/NBody/
  - http://shodor.org/petascale/materials/UPModules/NBodyScaling/

# Additional Suggested Exercises

- Investigate scaling is the number of bodies is increased (from say 10000 to 100000 or even 1000000). Does Parallelizing the position calculations begin to make a difference?
- Investigate the effect of scheduling (static, dynamic, etc) of the various loops on runtime.
- For a particular schedule type, investigate the effect of chunk size on run time.