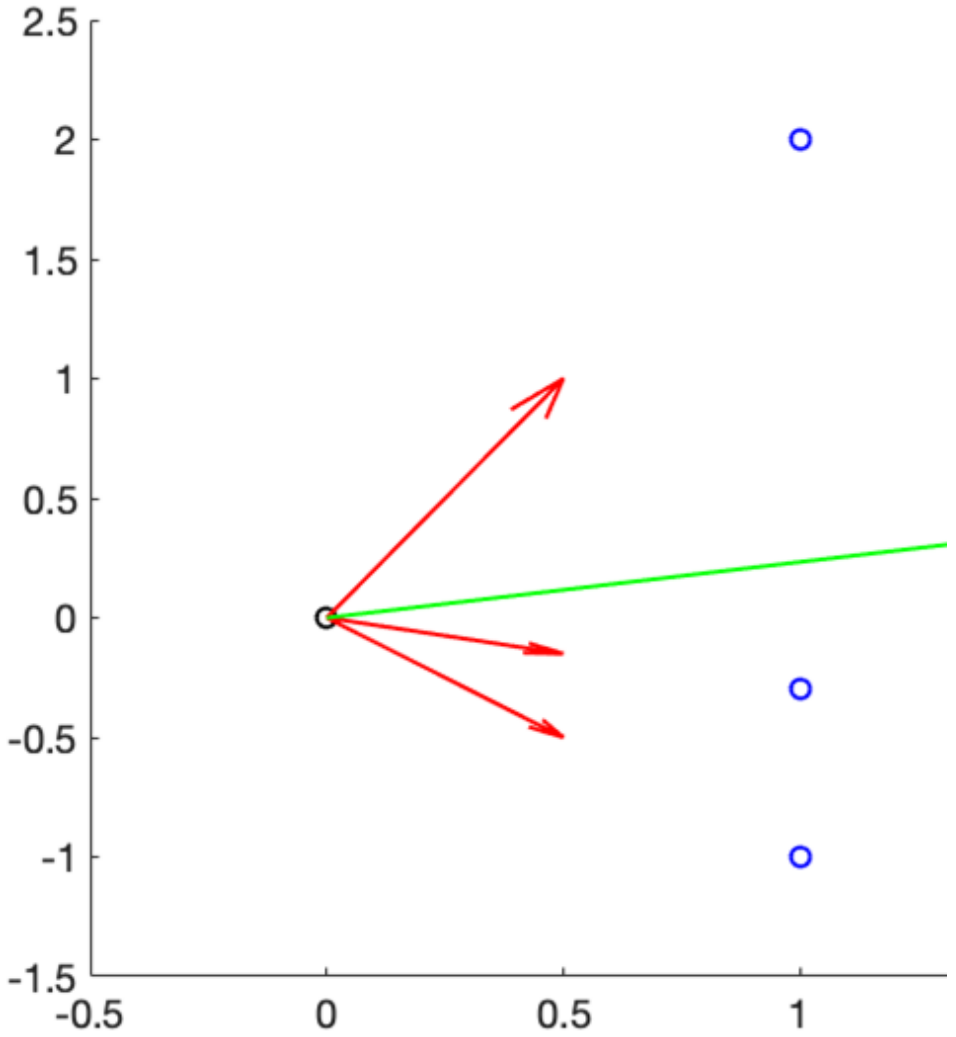
A simple N-Body Problem Parallelized with MPI

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, $(N^2\text{-}N)/2$ 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: $\overrightarrow{F_{i,j}} = \frac{Gm_i m_j}{|\overrightarrow{r_{i,j}}|^3}\overrightarrow{r_{i,j}}$ 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:

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

Deriving Verlet's Method (1/2)

-
- The displacement of the i[th] particle can be expanded forward in time as a Taylor series (to 3[rd] 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 Δt 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:

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

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:

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

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

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

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 coorindates Δt 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

How to Parallelize?

Strengths of MPI

- MPI works by passing messages back and forth between executables
- The various instances do not need to share the same physical memory
- MPI can allow us to expand our problem beyond the capabilities (physical memory) of one node

MPI at the beginning

- Need to set up the environment
- Each processor determines how many processors there are and its individual rank
- Also need to allocate gx, gy & gz as they will store all of the x, y & z coordinates at a given time index.
  - We want to allocate here, once in main()
  - We do not want to allocate with every call to accelerations()
- We also need to check the total number of bodies is divisible by the number of processes – if not some bodies might be missed in loops due to integer rounding!

```
int numprocs, rank;

// For storing x, y, & z positions of all
// the particles across all processor
// at a given time

double *gx,*gy,*gz;
gx=new double[GlobalNbodies];
gy=new double[GlobalNbodies];
gz=new double[GlobalNbodies];

MPI_Init(NULL, NULL);

// Get the number of processes/rank

MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);


// Check to make sure GlobalNbodies is divisible by
// numprocs – loops will fail if not!

  if (GlobalNbodies%numprocs){
       cerr << "GlobalNbodies not divisible by num of
                    processors!!!\n";
       return(GlobalNbodies%numprocs);}

// Setting Nbodies like this allows us to use the serial
main body largely intact
```

MPI in acceleration()

- As gx, gy, and gz (our arrays to hold all the x, y, and z coordinates at a particular time step) are allocated outside this function in the main code they need passed
- Collective communication (MPI_Allgather) is used to communicate the positions stored on each node, to every other node

```
void acceleration(double x[], double y[],
     double z[], int itime, int Nbodies,
     double mass, double ax[],
     double ay[],double az[], double gx[],
     double gy[], double gz[],int numprocs,
     int rank) {

// < . . . Snip . . .>

// MPI STUFF
// We need each processor to have a complete
copy of all the positions these three lines will
accomplish that

ibody1=0;
MPI_Allgather(&x[index(itime,ibody1,
     Nbodies)], Nbodies, MPI_DOUBLE, gx,
     Nbodies, MPI_DOUBLE,
     MPI_COMM_WORLD );
MPI_Allgather(&y[index(itime,ibody1,
     Nbodies)], Nbodies, MPI_DOUBLE, gy,
     Nbodies, MPI_DOUBLE,
     MPI_COMM_WORLD );
MPI_Allgather(&z[index(itime,ibody1,
     Nbodies)], Nbodies, MPI_DOUBLE, gz,
     Nbodies, MPI_DOUBLE,
     MPI_COMM_WORLD );
```

MPI in acceleration()

- Instead of looping over just the Nbodies stored on the system, we need one of the loops to loop over all the bodes, stored on all the systems.
- The inner loops were chosen as that results in the fewest code changes
- gx, gy, and gz need to be substituted for x, y, and z in the corresponding particle
- Additionally the equation for the force due to the center mass need to change slightly as well.

```
for(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)+1.0e8;
    // 1.0e8 is to soften the potential, prevent
    //        "explosions"
    F=(G*10*mass*Nbodies*numprocs/(r*r*r));
    // a central mass/star/something with the
    //         10x mass as the system
    ax[ibody1]=-F*dx;
    ay[ibody1]=-F*dy;
    az[ibody1]=-F*dz;
    for (ibody2=0;ibody2<Nbodies*rank+ibody1-1;
              ibody2++) {
         dx=x[index(itime,ibody1,Nbodies)]-gx[ibody2];
         dy=y[index(itime,ibody1,Nbodies)]-gy[ibody2];
         dz=z[index(itime,ibody1,Nbodies)]-gz[ibody2];
          r=sqrt(dx*dx+dy*dy+dz*dz)+1.0e8;
          // 1.0e8 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;}
     for (ibody2=Nbodies*rank+ibody1+1;
            ibody2<Nbodies*numprocs;ibody2++) {

< . . . Snip . . .>
```

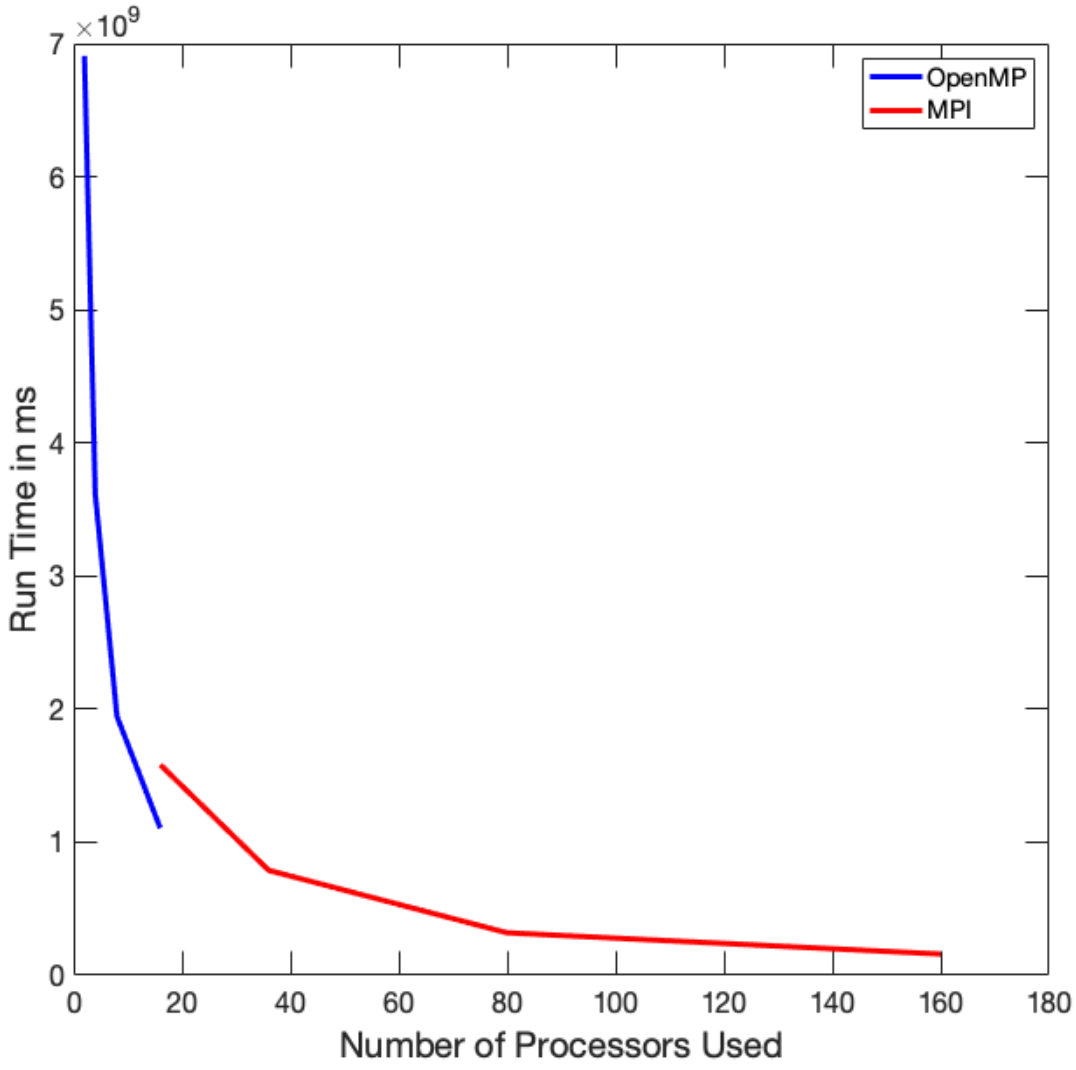
MPI IO

- First we need to open the files.
  - We are instructing the call to MPI_File_open to create the file if needed, and to open it write only.
  - Note the or “|” between the status variables
- The serial code writes the arrays out to binary files
- In order to maintain compatibility, we are using the collective call MPI_File_write_at_all to write to the file
- Not it needs not only what is to be written, but also an offset in bytes so it knows where each process should write to.
- MPI collective IO routines avoid unnecessary communication to a central process which then does the write.

```
MPI_File mpifilex, mpifiley, mpifilez;
MPI_Status status;
MPI_Offset offset;


MPI_File_open(MPI_COMM_WORLD,"x.bin",
   MPI_MODE_CREATE | MPI_MODE_WRONLY,
   MPI_INFO_NULL,&mpifilex);
MPI_File_open(MPI_COMM_WORLD,"y.bin",
   MPI_MODE_CREATE | MPI_MODE_WRONLY,
   MPI_INFO_NULL,&mpifiley);
MPI_File_open(MPI_COMM_WORLD,"z.bin",
   MPI_MODE_CREATE | MPI_MODE_WRONLY,
   MPI_INFO_NULL,&mpifilez);

for (itime=0; itime<Ntime;itime++){
      offset=sizeof(tempr)*
            (itime*GlobalNbodies+rank*Nbodies);
      MPI_File_write_at_all(mpifilex,offset,&x[itime*Nbodies],
            Nbodies,MPI_DOUBLE,&status);
      MPI_File_write_at_all(mpifiley,offset,&y[itime*Nbodies],
            Nbodies,MPI_DOUBLE,&status);
      MPI_File_write_at_all(mpifilez,offset,&z[itime*Nbodies],
            Nbodies,MPI_DOUBLE,&status); }

MPI_File_close(&mpifilex);
MPI_File_close(&mpifiley);
MPI_File_close(&mpifilez);
```

Runtime as a function of the number of processors

- We have included an OpenMP version (same as in the N-body OpenMP module) for comparison.
- Note that at 16 processors, the OpenMP code is about 30% faster! Communication takes time – shared memory is faster (this is why people write MPI/OpenMP Hybrid codes, to get the best of both!)
- MPI does allows us to expand significantly beyond the limitations of a single node – large SMP nodes get very expensive, quickly, thus most machines only have limited SMP ability



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

- If a problem gets too small, it will stop scaling earlier. Take the existing MPI code, how does it scale for a problem that is only $1/10^{th}$ the size (total number of bodies)
- How does the code scale for a problem 10 times larger?
- Rewrite the IO section such that all the data for a particular time is gathered to process 0 and then written with the usual C++ fwrite command. Is the IO faster or slower than MPI collective routines? By how much?