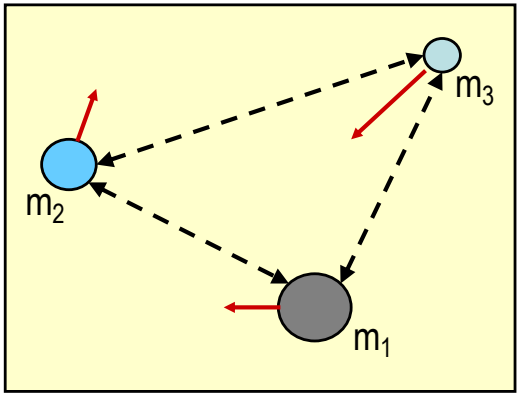
N-body problem

- N-body simulations are a class of computational problems where we calculate the effect of a force between *N* objects (or bodies)
- The problem is to calculate the positions and movements of a number of bodies in space as time advances
  - the bodies are affected by all other bodies via gravitation
  - long-range interactions: each object affects all other objects
- Here we will only consider the two-dimensional case
  - extension to three dimensions is straight forward
- We will only consider gravitational force
  - attraction between two bodies
  - can also use the same methods for other types of forces, for instance electrostatic attraction or repulsion

Problem description

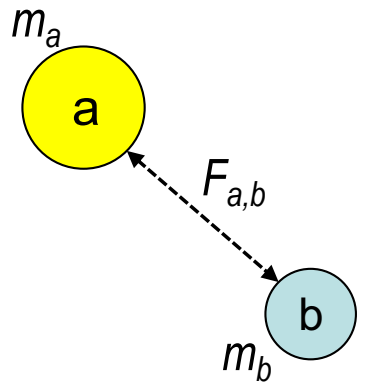
- *We have N* bodies in 2-dimensional space
  - bodies are treated as point masses
  - shape and size does not affect behaviour



- Each body is described by its
  - mass *m*
  - position $X = (x_x, x_y)$
  - velocity $V = (v_x, v_y)$
    the rate of change in position over time, $V = dX/dt$
  - acceleration $A = (a_x, a_y)$
    the rate of change in velocity over time, $A = dV/dt$

- As a body is affected by a force, its velocity changes
  - Newton's laws describe how bodies in space affect each other with gravitation

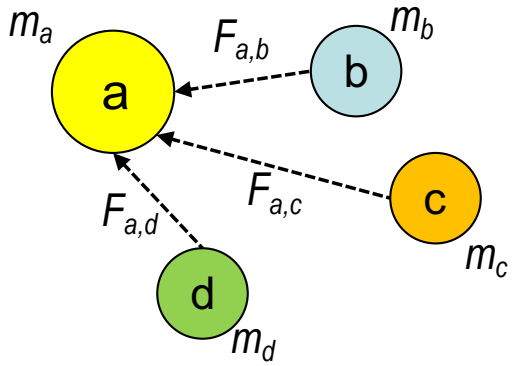
Laws of gravitation



- Newton's laws
  - two bodies *a* and *b* with masses m_a and m_b
  - positions are $X_a = (x_a, y_a)$ and $X_b = (x_b, y_b)$
  - gravitational force on *a* caused by *b*
    - $F_{a,b} = \frac{G m_a m_b}{r^2} \quad \frac{X_b - X_a}{r}$ where *G* is the gravitational constant, G = 6.67259e-11
    - and $r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$ is the distance between *a* and *b*
  - forces are vectors with *x*- and *y*-components $F = (f_x, f_y)$
  - pairwise forces are symmetric but of opposite direction: $F_{a,b} = -F_{b,a}$

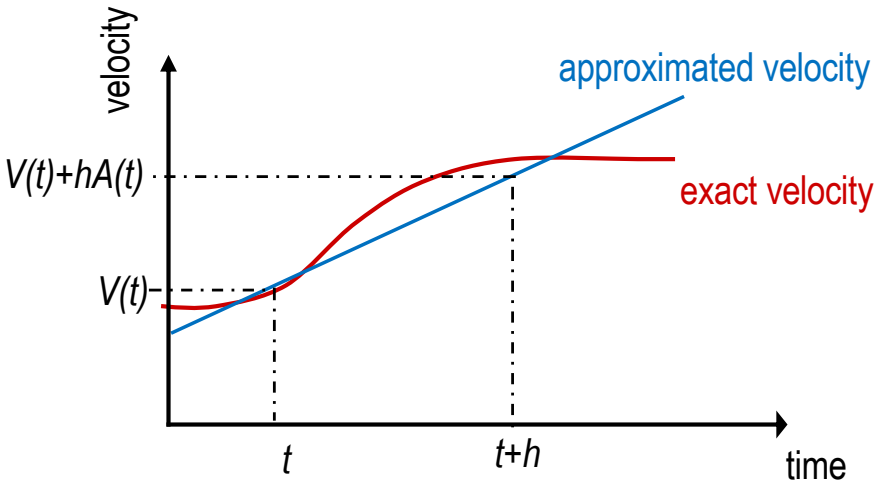
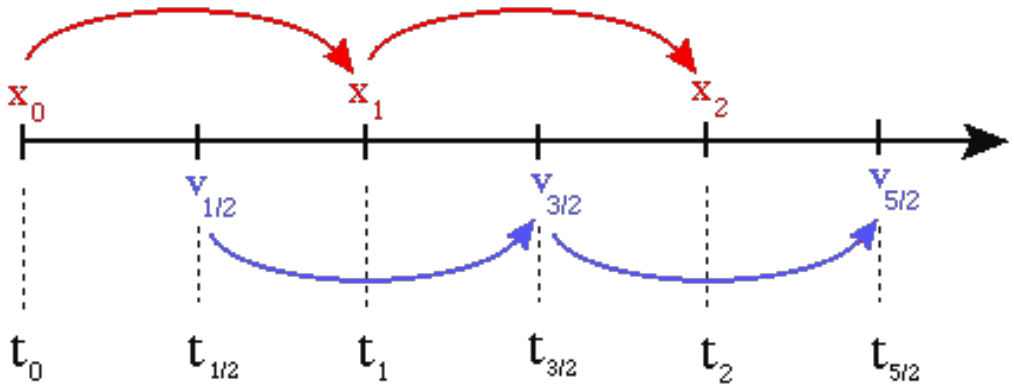
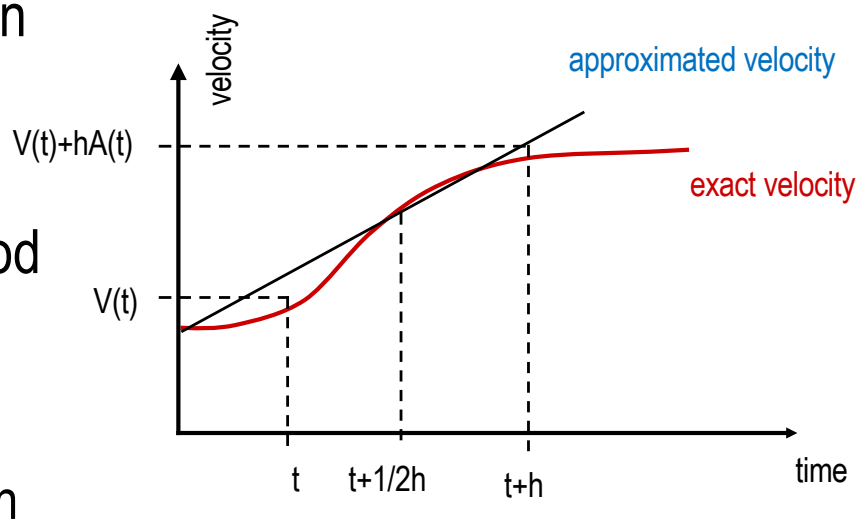
Laws of gravitation (cont.)

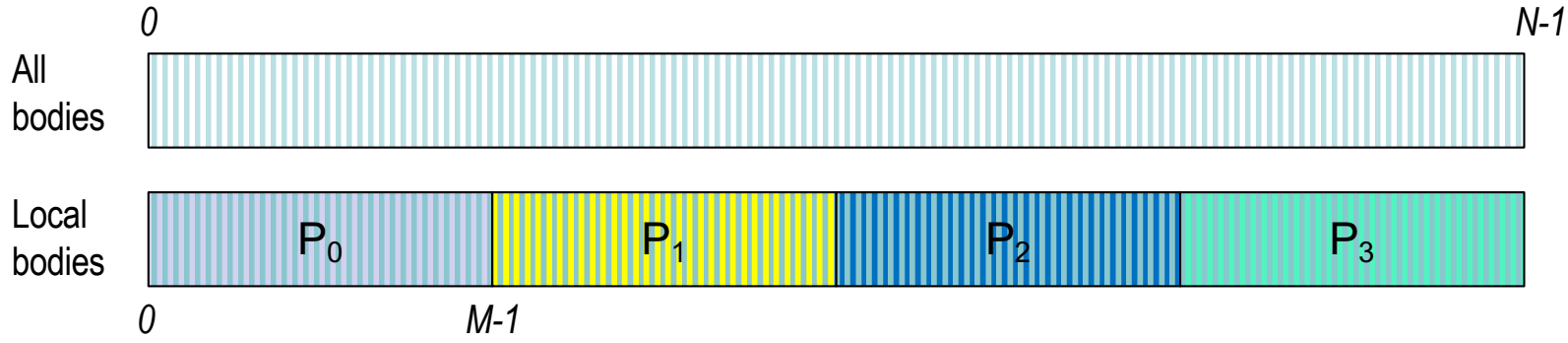
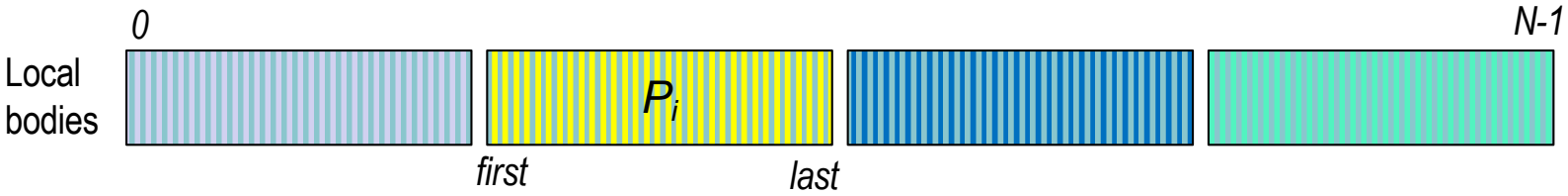
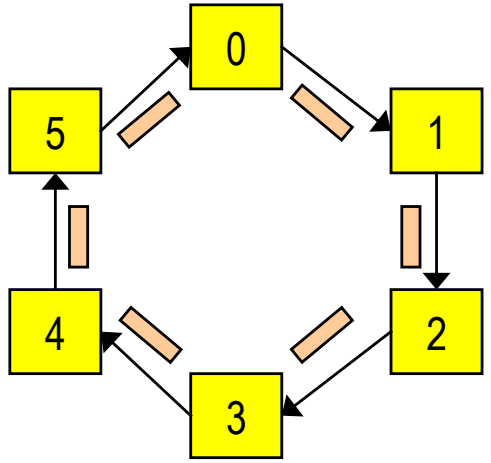
- The total force on a body *a* is the sum of the pairwise forces from all the other bodies
  - $F_a = \sum_{j=0, j \neq a}^{N-1} F_{a,j} = G\, m_a \sum_{j=0, j \neq a}^{N-1} m_j \left[\frac{X_j - X_a}{r_{a,j}^3} \right]$



- The force on a body affects its motion according to Newton's second law
  $F = m * a$
  - $a = F/m$ (acceleration is force divided by mass)
- Given the current position of a body (x_a, y_a) and the acceleration we can compute the velocity and position of the body in the next time step

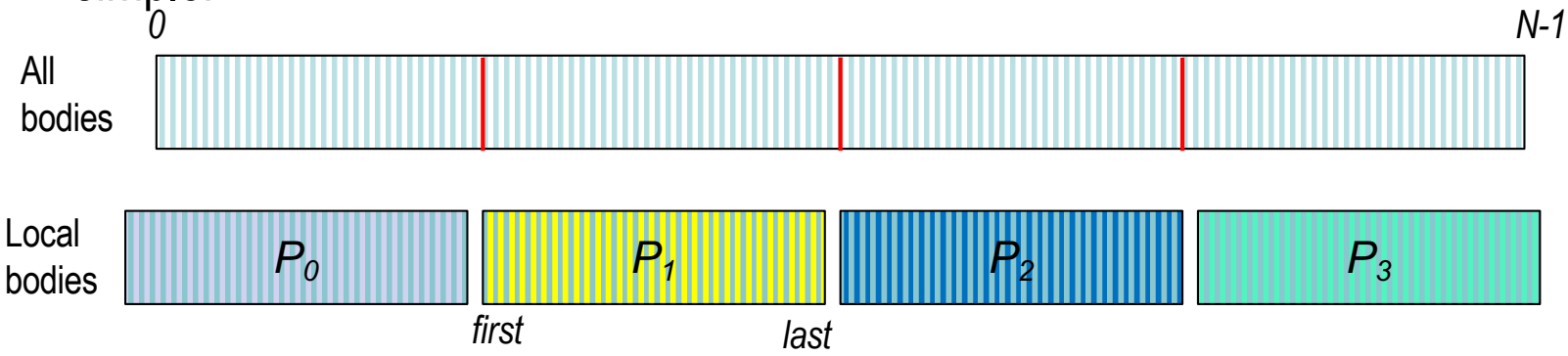
Discrete solution

- We divide the time into short time intervals of length h (often denoted Δt)
  - starting from an initial state at time t_0, we calculate the position and velocity for each body at times t_1, t_2, t_3, ...
  - if the current timestep is t_i then the next timestep is $t_{i+1} = t_i + h$
- $O(N^2)$ algorithm
  - ```
    for each time interval
      for each body b
          for all other bodies c
              calculate the force on b caused by c
          calculate new velocity for b
          calculate new position for b
    ```
- The time interval h has to be short enough to give an accurate solution
  - the amount of computation increases with a shorter interval
    ```

Discrete formulation

- For a body with mass *m* we compute the force *F* affecting it at time *t*
  - sum of the forces from all other bodies
- Then we compute the velocity and position of the body for the next timestep, i.e. at time *t*+*h*
  - new velocity is $V^{t+1} = V^t + A^t h$ where $A^t = \frac{F}{m}$
  - new position is $X^{t+1} = X^t + V^t h$
- When the bodies move to new positions, the forces change and the computation has to be repeated
- Called Euler's method

Problem with Euler's method

- In Euler's method, we assume that the acceleration is constant during the time interval
  - we use the acceleration at time *t* to calculate the new velocity at time *t+h*
- Acceleration is not constant during a time interval
  - when a body gets closer to another body, its acceleration increases
  - our approximation is based on the velocity at time *t*
- Have to use a small time step to get reliable results



The Leapfrog method

- In the Leapfrog method we use the *mid-point* of the interval to approximate the velocity during the interval
  - this is used to calculate the positions at the next point in time
- Velocities and positions are not updated at the same point of time
  - update positions at the *beginning* of each time step and velocities at the *middle* of the time steps
- Positions are updated at times *h, 2h, 3h, …*
  - $X_{i+1} = X_i + h*V_{i+1/2}$ for *i=0,1, 2, ….*
- Velocities are updated at times *1/2h*, *t+1/2h, 2t+1/2h, 3+1/2h, …*
  - $V_{i+1/2} = V(t+1/2h)$



Properties of the Leapfrog method

- Using the velocity at the midpoint of the interval gives a better approximation than the velocity at the beginning (or end) of the interval

- No more complicated to implement than the Euler scheme
- The initial velociy at $V_{1/2}$ can for instance be calculated by Eulers method



- The leapfrog method is time reversible
  - starting from any state at time t_i we can calculate backwards in time to t_0
- The leapfrog method is a second order approximation
  - the approximations of the positions have an accuracy of $O(\Delta t^2)$

Implementation of the Leapfrog method

- A sequential implementation of the Leapfrog method is

```
set initial values of V to the velocities at time 0.5*deltat

for (t=0; t<maxt; t++) {          /* For each timestep */
   for (i=0; i<nbodies; i++) {     /* For each body */
      X[i] = X[i]+V[i]*deltat;        /* Update position */
      F = Force(i);                   /* Calculate force on i */
      V[i] = V[i]+F*deltat/m;         /* Update velocity */
   }
}
```

- First update positions, then forces and acceleration and finally velocities
  - no need to keep positions and velocities from the previous timestep
  - first move the bodies based on their current velocities
  - then calculate the new forces based on the new positions
  - finally, calculate the new velocities

Parallel *N*-body simulation: decomposition

- *N* bodies, *P* processes
  - we divide the bodies evenly among the processes
  - *M* = *N/P* local bodies per process



- A process updates the positions, forces and velocities *for its own bodies*
  - to calculate the forces, it needs information about the positions of *all the other* bodies
- The algorithm is the same as in a sequential solution, but it is only applied to the *own subset of bodies*
  - move the bodies, exchange updated positions with all other processes
  - calculate the forces on own bodies
  - calculate the new velocities for the own bodies

Parallel *N*-body simulation: updating the bodies

- Each processes stores and updates its own range of the particles
  - each process has *M* = *N/P* local bodies
  - can use variables *start* and *end* to store the interval of bodies for each process
    - *first* = *N*id/np*
    - *last* = *N*(id+1)/np*

    where *N* is total number of bodies, *id* is the process rank and *np* is the number of processes



- Move the bodies to new positions:
  - similar code to update velocities

```
for (int i=first; i<last; i++){
      X[i] = X[i] + Vx[i]*dt;
      Y[i] = Y[i] + Vy[i]*dt;
}
```

- Calculate forces for the updated positions:

```
ComputeForce_parallel(first, last, N, X, Y, mass, Fx, Fy);
```

Communication structures

- A parallel implementation of the N-body simulation requires all-to-all communication
- There are different ways to implement the communication
  - can use MPI collective communication: broadcast / scatter / gather / all-to-all
  - can use point-to-point communication to implement data exchange between all processes
- Using collective communication leads to shorter code and (perhaps) a clearer program structure
  - requires a good understanding of how collective communication works
- Using point-to-point communication may be more familiar and is based on simple communication primitives
  - can use same ideas as in the ring communication exercise
  - processes are arranged in a ring where the local bodies are circulated among the processes

Communication structure: ring of processes

- All-to-all communication implemented in a ring of processes
- The positions of the bodies are circulated through a ring
  - a process P_i updates the positions of its own local bodies and sends them to the next process in the ring
  - P_i calculates the forces between its own local bodies
  - when P_i receives the positions from some other process P_j, it calculated the force these cause on its own bodies
  - after *P-1* communication steps, all bodies have visited all processes and the total forces have been calculated
  - then the processes can update new velocities for the next timestep



- Communication can proceed simultaneously between all processes
  - all processes have the same amount of work, both computation and communication

Algorithm structure

- Algorithm for a ring-based solution
  - `initialize all velocities at time 0.5*deltat`
  - ```
    for each time interval
      update positions of local bodies
      calculate forces between the own local bodies
      for all other processes
          send local bodies to next process in the ring
          receive bodies from previous process in the ring
          compute forces between local bodies and received bodies
      update velocities of local bodies
    ```

- The force values are accumulated with contributions of particles both from local bodies and from the other processes
  - all these are added together to get the total forces on the local particles
    ```

Implementation with collective communication

- We can also use MPI collective communication to exchange information about the bodies
  - the communication does not need to do the data exchange in *P-1* steps, but can directly send the needed data to all processes
  - the implementation is otherwise similar, but the communication structure can be simpler



- The exchange of particle positions is a gather-operation
  - coordinates of the local bodies in each process are gathered into one array
  - the result needs to be distributed to *all processes*
- Can implement the data exchange with an *MPI_Gather* followed by a *MPI_Broadcast* of the result
  - or we can use an *MPI_Allgather*

MPI_Gather and MPI_Allgather

- MPI_Gather collects the result into *recvbuf* on the *root*-process
  - `int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
- MPI_Allgather has no root-process, the results is in *recvbuf* in all processes
  - `int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
- Can use *MPI_IN_PLACE* as the value of *sendbuf* on the *root*-process
  - the gather-operation is done in-place, i.e. *recvbuf* is also used as *sendbuf*
  - values of *sendcount* and *sendtype* are ignored, the receive-parameters are used to also determine send-parameters
- Wnen *MPI_IN_PLACE* is used, *sendcount* can be given as 0 and *sendtype* as *MPI_DATATYPE_NULL*

Testing the implementation

- Test the parallel implementation with the same input data as in the sequential program
  - if you don't change the random number generator or the seed values, the same initial positions and masses will be generated
- The program writes out the final positions of the bodies
  - compare these to the result from the sequential program
  - in a correct parallel implemetation, they should be the same at least with 2 decimals of precision
- You can for instance use the Unix command *diff* to compare two files
  - *diff file.txt file2.txt* compares the two text file and prints out the rows that differ
  - *diff –y file1.txt file2.txt* writes out the files in two columns side by side
  - see the manual page for *diff* (*man diff*)