Module 5. Particle Applications – Pipeline Computing

Many questions in science can be answered by viewing a physical system as a collection of particles that obey certain laws. A familiar example is that the universe can be viewed as a collection of astronomical bodies which obey Newton's laws of gravitation.

Examples

*	*the solar system*
*	*an electrical or electronic network*
*	*a globular star cluster*
*	*the atoms in a crystal vibrating under interatomic forces*
*	*a molecule rotation and flexing under interatomic forces*

The laws in the above examples can be written as equations that are typical of the class of equations known as Ordinary Differential Equations (ODEs). These equations have well-known solution techniques which can be easily expressed in a data parallel way. In the case of particle systems, solving the equations also involves calculating a function involving interactions between all pairs of particles. We will discuss a variety of ways to approach such "all pairs" calculations in a data parallel way, as well as discuss such issues as deciding which parts of a calculation to parallelize and how to achieve load balancing in a program.

5.1 Particle applications

The application that we will discuss is the universe of astronomical particles under Newton's laws of motion, an example of what is commonly known as the N-body problem. We suppose that our system consists of N particles, each with a mass, moving with some velocity through three-dimensional space. Part of Newton's system is the recognition that mass and velocity of the particles are what affect the system; in particular, we can disregard the diameters and shapes of the particles and treat them as *point masses*.

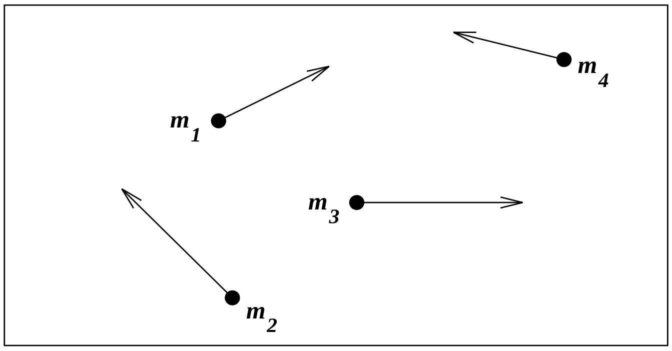


Figure 1: Particles at one point in time with velocity vectors and masses.

Velocity is, of course, defined to be the change in position over time, and acceleration to be the change in velocity over time.

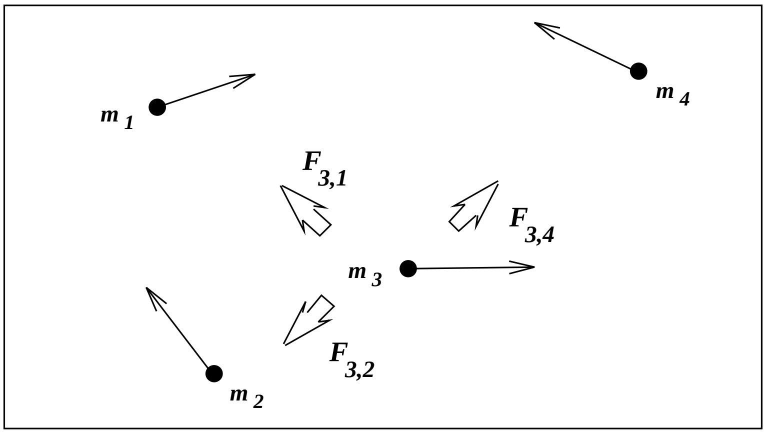


Figure 2: The force on particle 3 due to the other particles, shown in attractive form.

For all particles P_i with mass m_i, where $1 \leq i \leq N$,

position	$X_i = < x_i\ ,\ y_i\ ,\ z_i >$	*(3-D space)*
velocity	$V_i = \frac{dX_i}{dt}$	
acceleration	$A_i = \frac{dV_i}{dt}$	

Newton's equations of motion describe how position, velocity, and acceleration are related for astronomical bodies. Newton's First Law, the Law of Gravition, defines the force on one particle in the system due to the other particles.

Law of gravitation—the force on particle i due to particle j:

$$F_{ij} = G * m_i * m_j * \frac{X_j - X_i}{r_{ij}^3}$$

where G is the gravitational constant,

and r_{ij} is the Euclidean distance,

$$|X_j - X_i| = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2}$$

Newton's Second Law relates this force to motion:

$$F = m * A$$

Assuming that the masses are fixed, from these laws we can derive equations that express how position and velocity change over time.

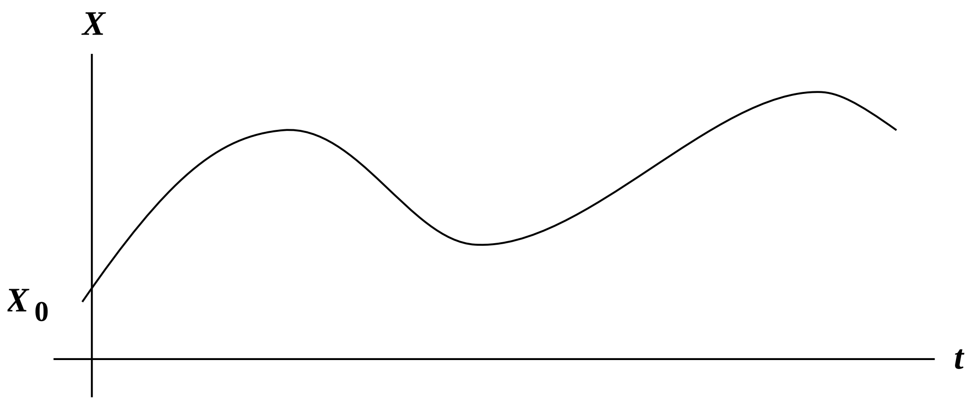


Figure 3: The equation for the derivative of position X over time t and an initial value for X determines a function of X and t.

Equations of Motion:

For N particles, let $X = [X_1, \ldots, X_N]$, $V = [V_1, \ldots, V_N]$, $A = [A_1, \ldots, A_N]$

$$\begin{aligned} \frac{dX}{dt} &= V \\ \frac{dV}{dt} &= A\,, \quad \textit{where} \quad (\forall 1 \le i \le N) \qquad A_i = G * \sum_{j=1\,,\,j\neq i}^{N} m_j \left(\frac{X_j - X_i}{r_{ij}^3} \right) \end{aligned}$$

Notice that the equation for acceleration is expressed in terms of position, so that we can represent the current state of a system of particles by the positions and velocities of the particles. From these equations, the positions and velocities of the particles can be evolved over time from an initial state.

Example question

> *Given positions and velocities of the sun, planets, their major satellites, and the asteroids, what will be their motions for the next 500 years? Will any asteroids crash into the earth during that time?*

That is, given an initial position, each equation determines a function of time.

The situation is entirely analogous for molecular dynamics, except for the nature of the forces on the particles. Instead of the inverse square law of Newtonian gravity, we might have a force law such as

$$F_{ij} = m_i * m_j * (X_j - X_i) * \left(\frac{A}{r_{ij}^6} - \frac{B}{r_{ij}^{10}} \right)$$

where A and B are the constants representing the attractive and repulsive forces, respectively, between the particles.

5.2 Numerical techniques for solving Ordinary Differential Equations

The equations of motion described above are an example of a system of Ordinary Differential Equations (or ODEs, as it is commonly abbreviated). Such systems of equations may be very difficult to solve analytically, for example, the equations of motion involve integrating over a sum. So typically, a numerical approximation is made for the integral by using the equation to approximate the change over time.

The equations that we wrote down above were also all *first order*, involving only first derivatives. Furthermore, if we had written down a second-order equation, such as expressing acceleration as the second derivative of position,

$$A = \frac{d^2X}{dt^2},$$

this equation could always have been rewritten as two first order equations by introducing the first derivative quantity, in this case, velocity. So we need only consider systems of first order equations, and, for simplicity, we will discuss only one equation. So in general we have an equation

$$\frac{dX}{dt} = f(t, X),$$

where f is used for a function expressing the right hand side of the equation.

The kinds of problems that we mentioned are called *initial value problems*, in contrast to *boundary value problems*, in which values for X are given at two end points and which require slightly more complicated techniques than will be discussed here.

Runge-Kutta Methods

We can solve an initial value problem by numerical approximation by setting up a grid of time steps. Starting from the initial value, we can then calculate positions one step at a time, using the equation for the derivative of the function to calculate the change in the function over one small time step.

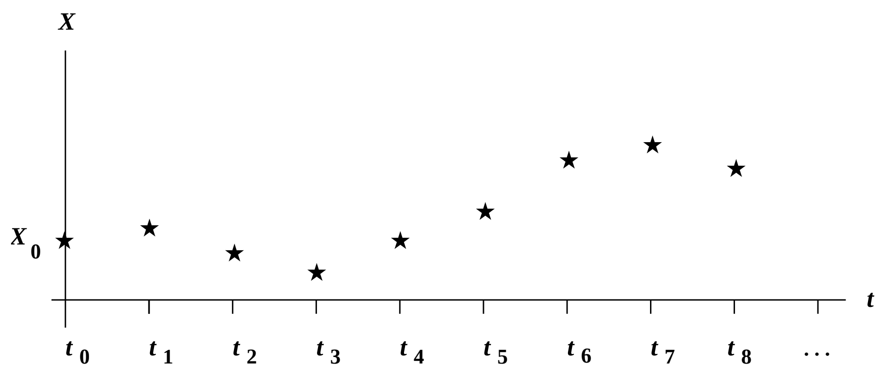


Figure 4: Calculate the function X of t one step at a time using its derivative $f(t, X)$.

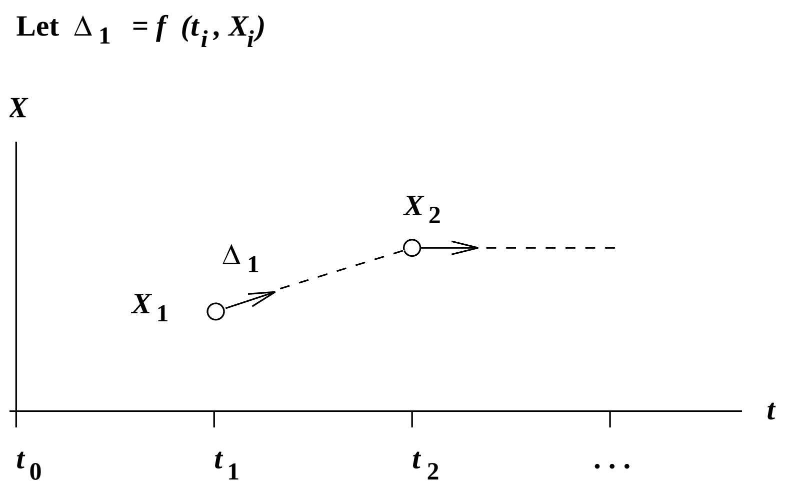


Figure 5: Use the derivative at one time step to extrapolate the function value at the next time step.

A simple example of a technique for finding these values of X is given by *Euler's Method*. This method is not accurate enough for practical use, but serves as a good illustration of these kinds of techniques. We first pick a time step h, which will be a uniform interval between the t_i's.

Euler's method:

Choose $h > 0$ *to be the step size. Then*

$$t_i = t_0 + h * i \,, \quad \text{for} \quad 0 \leq i \leq \frac{(t_{last} - t_0)}{h}$$

Approximate solutions are given by

$$\boxed{\begin{array}{rcl} X_0 & = & \textit{initial value} \\ X_{i+1} & = & X_i + h * f\,(t_i\,, X_i) \end{array}}$$

This method uses the value of the function X_i and the derivative at that point in time $f\,(t_i\,, X_i)$ to make a simple linear approximation X_{i+1} of the value of the function at the next point in time. In fact, whenever f satisfies certain smoothness conditions, there is always a sufficiently small step size h such that the difference between the real function value at t_i and the approximation X_{i+1} is less than some required error magnitude e. [Burden and Faires]

Although Euler's method can produce good function approximations in theory, it is not used in practice because of the amount of computation required to produce a specified error e is large compared to the other methods. In all these methods, the assumption is that the derivative function f is a complicated expression requiring substantial computation (otherwise analytical methods would be superior). Notice that in Euler's method, f is computed once for each step. Smaller step sizes therefore require correspondingly more computations of f over the entire interval.

In order to compare Euler's method with other more practical ones, we can analyze the order of the error in terms of the step size. Suppose that Y_i is the exact solution at time t_i. Then the approximation over one step size can be represented as follows.

Use Taylor's theorem to represent the exact solution:

$$Y_{t_i+h} = Y_{t_i} + h * \frac{dY_{t_i}}{dt} + \frac{h^2 d^2 Y_{t_i}}{2!\ dt} + \ldots + \frac{h^p d^p Y_{t_i}}{p!\ dt} + \ldots$$

Use the O notation to represent the remaining terms in higher derivatives, of which the dominant term is h^2.

$$Y_{t_i+h} = Y_{t_i} + h * \frac{dY_{t_i}}{dt} + O\left(h^2\right)$$

Assume X is exact at t_i:

$$Y_{t_i+h} = X_{t_i} + h * \frac{dX_{t_i}}{dt} + O\left(h^2\right)$$

The derivative is given by the original equation to be solved:

$$Y_{t_i+h} = X_{t_i} + h * f\left(t_i\,, X_i\right) + O\left(h^2\right)$$

Therefore,

$$Y_{t_i+h} = X_{t_i+h} + O\left(h^2\right)$$

So the difference between the exact solution and the approximate solution at one time step, the *local error*, is $O\left(h^2\right)$. Over the entire interval, however, this multiplies to obtain *global error* of $O(h)$. This method is called a *first order method*, corresponding to the exponent of h in the global error.

An improvement of this method is called the *modified Euler method* or the *midpoint method.* In this method, the derivative at one time step is used to estimate a value of the function at the mid-point of the time step. This function value is not used, but its derivative is calculated and used to obtain a better estimate of the function value at the next time step. This method corresponds to using Taylor's theorem up to the h^3 term.

Modified Euler method:

$$
\boxed{
\begin{array}{rcl}
X_0 & = & \textit{initial value} \\
\Delta_1 & = & f\left(t_i\,,X_i\right) \\
\Delta_2 & = & f\left(t_i + \frac{1}{2}h\Delta_1\,,X_i + \frac{1}{2}h\Delta_1\right) \\
X_{i+1} & = & X_i + h * \Delta_2
\end{array}
}
$$

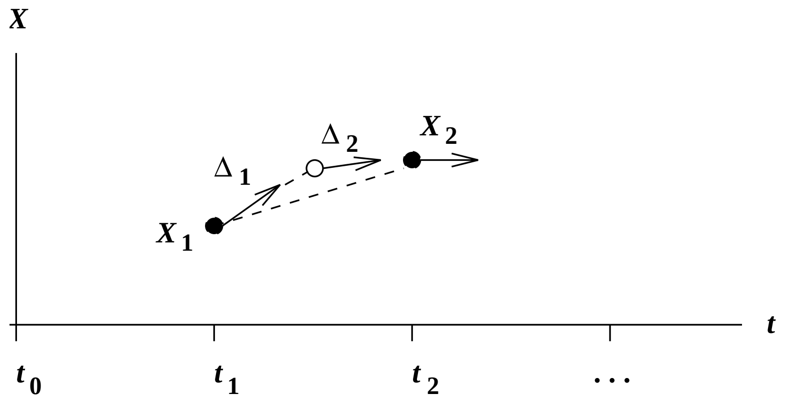


Figure 6: Use the derivative at one time step to extrapolate the midpoint values; use its derivative to extrapolate the function value at the next time step.

This method computes the derivative function twice at each time step, but achieves second-order accuracy.

The *classical Runge-Kutta method* extends these ideas further to not only obtain derivatives of midpoints but the derivative of a trial endpoint. The advantage of this is that we can then use more accurate integration techniques to obtain the endpoint. That is, we can use the following equation to find X_{i+1}.

$$X_{i+1} - X_i = \int_{t_i}^{t_{i+1}} f(t, X)$$

Using Simpson's rule for this integral gives us the desired accuracy as long as the three derivatives at the intial point, the midpoint and the endpoint have that accuracy as well.

The derivative of the first point of the time step is used to obtain a trial midpoint. The derivative of this point is used from the beginning to obtain a second trial midpoint. Its derivative is used to obtain a trial endpoint of the time step. Finally, the values of the derivatives at the initial point, the average of the midpoints, and the endpoint are integrated using Simpson's rule to obtain an endpoint which will be used as the function value over the next time step.

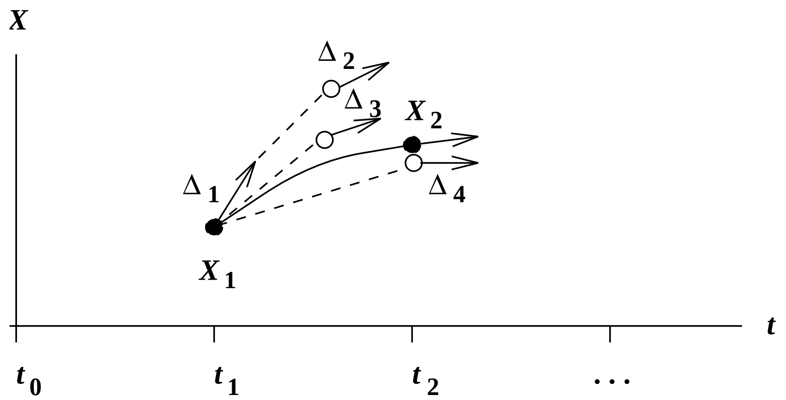


Figure 7: Fourth order runge-Kutte with two trial midpoints, a trial endpoint, and the final function approximation X_2.

Classical Runge-Kutta method:

$$
\boxed{
\begin{array}{rcl}
X_0 & = & \textit{initial value} \\
\Delta_1 & = & f\left(t_i\,, X_i\right) \\
\Delta_2 & = & f\left(t_i + \frac{h}{2}\, X_i + \frac{h}{2}\Delta_1\right) \\
\Delta_3 & = & f\left(t_i + \frac{h}{2}\, X_i + \frac{h}{2}\Delta_2\right) \\
\Delta_4 & = & f\left(t_i\,, X_i + h\Delta_3\right) \\
X_{i+1} & = & X_i + \frac{h}{6}\left(\Delta_1 + 2\Delta_2 + 2\Delta_3 + \Delta_4\right)
\end{array}
}
$$

Again, we will not show the full error analysis here, but the derivation of the midpoints and Simpson's rule at each time step leaves terms of $O\left(h^5\right)$. So the global error of Runge-Kutta is $O\left(h^4\right)$ and is achieved by computing four of the derivative functions f at each time step. Therefore, in general, the Runge-Kutta method does four times as much calculation at each time step but should only be required to do 1/16 as many time steps as the Euler method.

Other Methods

Another widely used class of methods are the *predictor/corrector methods*. In the last section, we saw that the Runge-Kutta methods used only the last known function value X_i and other values in the current time step in order to compute the next function value X_{i+1}. These methods are called *one-step methods*. In contrast, the predictor/corrector methods are *multi-step methods*; they first *predict* a function value for X_{i+1} and then substitute that approximate value into an implicit equation using several of the latest known function values to interpolate a function, giving a *corrected* value for X_{i+1}. These methods can be formulated at various orders of accuracy—an example would be the *fourth order Adams-Bashforth/Adams-Moulton* *method*—but are slightly more complicated to program as old function values must be saved for some number of iterations.

Many other methods are available and several references are given at the end of the chapter. It is not precisely known, however, which methods are best for a particular system of equations and experimentation is usually required to find the method which achieves the desired error for the least computation.

5.3 Computational solution

Returning to our Newtonian gravitation problems, we can now solve the equations of motion for an astronomical particle system like the solar system. We can then use our solver to start the particles in an initial state and end with their positions and velocities after some period of time, say 100 years. Recall our formulae:

Equations of Motion:

For N particles, let $X = [X_1, \ldots, X_N]$, $V = [V_1, \ldots, V_N]$, $A = [A_1, \ldots, A_N]$

$$\begin{aligned} \frac{dX}{dt} &= V \\ \frac{dV}{dt} &= A = Grav\,(X)\,, \quad where \quad (\forall 1 \leq i \leq N) \\ & \quad A_i = G * \sum_{j=1 j \neq i}^{N} m_j \left(\frac{X_j - X_i}{r_{ij}^3} \right) \end{aligned}$$

In the second equation, we introduce the name *Grav* for the derivative function.

The standard solution methods discussed in the last section all had the form of an iteration over time. In this context, we are using the subscript notation, for example X_i, to indicate values of the i^{th} particle, in this case position. Therefore, we shall use functional notation, for example $Y(i)$, to indicate the i^{th} step in the iterative solution.

Form of Solution:

$$\begin{aligned} Y(0) &= initial \;\; value \\ Y(i+1) &= Y(i) + h * (an \;\; expression \;\; in \;\; f, Y, and \;\; t) \end{aligned}$$

Disregarding the initial values for $X(0)$ and $V(0)$, the iterative solution for the equations of motion using Euler's method then become:

Euler's method:

$$\begin{aligned} X(i+1) &= X(i) + h * V(i) \\ V(i+1) &= V(i) + h * Grav\;(X(i)) \end{aligned}$$

Although this is not a practical computational method, its form is representative of the other methods like Runge-Kutta and the simplicity of the form allows us to more easily see the parallel form of the computation.

First of all, the computation involving the numerical method is inherently iterative. At each time step, the solution depends on the immediately preceding one. Then, if n is the number of time steps to solve the equations over the required interval, this part of the computation is $O(n)$.

In addition, at each time step, the function $Grav\,(X)$, must be computed. For each particle i, the total force is calculated, which involves summing over the forces on i due to all other particles j. So if there are N particles, each sequential computation of $Grav$ is $O\left(N^2\right)$. So a significant portion of the total computational effort is in calculating the derivative function.

The positions and velocities, $X = [X_1, \ldots, X_N]$ and $V = [V_1, \ldots, V_N]$, of the particles can be represented as arrays. Since each position X_i is actually represented by coordinates x, y, and z, in three-dimensional space, the positions and velocities are represented by $3xN$ arrays.

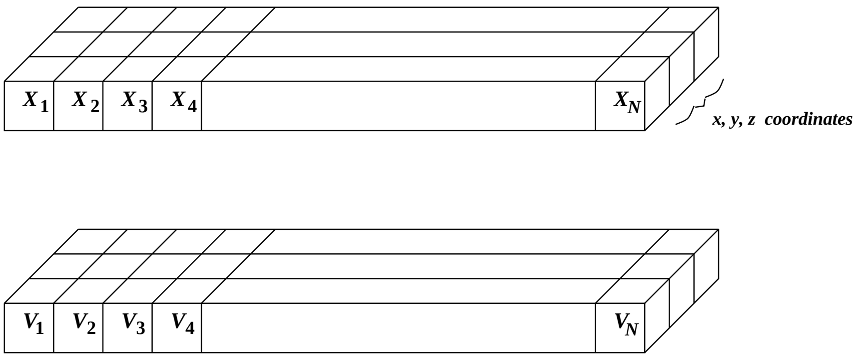


Figure 8: Arrays for positions and velocities at one time step.

The computation of the numerical method can be accomplished by a subroutine which takes the current positions and velocities, X and V, the masses M, a step size h, and a number of time steps, ns. It will leave the vectors X and V updated with the new state. The basic numerical method subroutine assumes that it can call a function *Grav* to compute the new accelerations. This function will be discussed later.

Now for each particle i, the computation of *Grav* requires summing over a set of terms, where each term uses position information from particle i and position information and the mass from particle j. So we can spread the position information into two $N \times N \times 3$ matrices, where for each pair of indices i and j, one matrix has position information of particle i and the other for particle j.

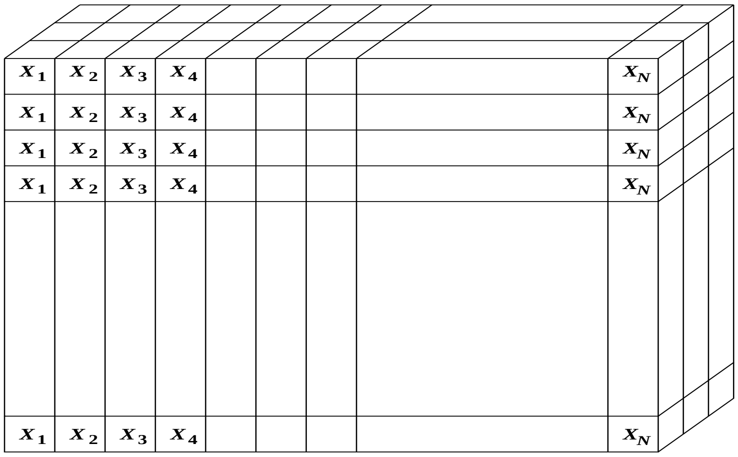
```
    subroutine runge-kutta (X, V, M, h, ns)
C Solves Newton's equations of motion using Runge-Kutta method
C which is globally 4th order. X and V are the initial positions and
C velocities. The system is evolved for a time interval h*ns.
C X and V contain the updated state at that time.

        real, dimension (1:3, 1:N) :: X,V, Xdelta1, Vdelta1, Xdelta2, Vdelta2,
     *                              Xdelta3, Vdelta3, Xdelta4, Vdelta4
        real h, h2
        integer ns, k
        INTERFACE
         function Grav(X,M)
         real, dimension(1:3, 1:N) :: X,M,Grav
         end function
        end interface

        h2=.5*h                        ! h/2, for convenience
        do k=1, ns
         Xdelta1 = V
         Vdelta1 = Grav (X,M)
         Xdelta2 = V + h2 * Vdelta1
         Vdelta2 = Grav (X + h2 * Xdelta1,M)
         Xdelta3 = V + h2 * Vdelta2
         Vdelta3 = Grav (X + h2 * Xdelta2,M)
         Xdelta4 = V + h * Vdelta3
         Vdelta4 = Grav (X + h * Xdelta3,M)
         X = X + h/6. (Xdelta1 + 2 * (Xdelta2 + Xdelta3) + Xdelta4)
         V = V + h/6. (Vdelta1 + 2 * (Vdelta2 + Vdelta3) + Vdelta4)
        end do
    end subroutine runge-kutta
```

So two spreads of the position array X and one spread of the masses m leaves all the particle information in position to calculate the terms of the sum using array operations. In actually computing the acceleration between two particles, for each particle we compute the equation:

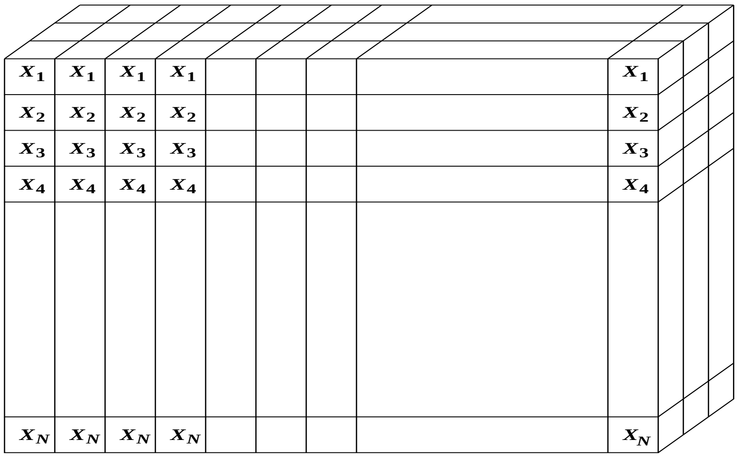
$$G * m_j \left(\frac{X_j - X_i}{r_{ij}^3} \right)$$

Let $D = X_j - X_i$ be the difference in positions. If D has coordinates $\langle x_D , y_D , z_D \rangle$, then $r_{ij} = \sqrt{x_D^2 + y_D^2 + z_D^2}$. In the arithmetic that follows, we must divide the three coordinates of D by r_{ij} and multiply them by $G * m_j$. The main diagonal can then be zeroed, since the formula specifies not using the terms where $i = j$, and the sum of the forces can be calculated by a sum reduction along the columns, yielding a new accelerations.

particles $\boldsymbol{X_i}$



particles $\boldsymbol{X_j}$



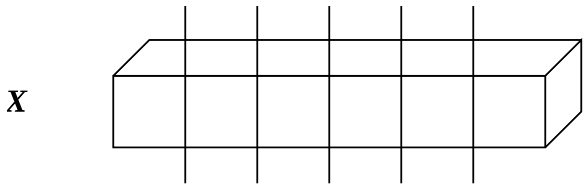
masses $\boldsymbol{m_j}$

m_1	m_1	m_1	m_1					m_1
m_2	m_2	m_2	m_2					m_2
m_3	m_3	m_3	m_3					m_3
m_4	m_4	m_4	m_4					m_4
m_N	m_N	m_N	m_N					m_N

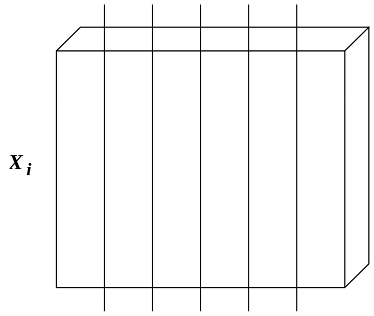
$$Grav(X) : (\forall 1 \leq i \leq N) G * \sum_{j=1,\, j \neq i}^{N} m_j \left(\frac{X_j - X_i}{r_{ij}^3} \right) \; where \;\; r_{ij} = |X_j - X_i|$$

```
    function Grav (X, M)
C accepts positions of particles X and masses of particles M
C returns accelerations in Grav
C Uses completely parallel calculation, ignoring anti-symmetry of
C force.
         integer, parameter :: N = number of particles
         real, parameter :: G = gravitational constant
         real, dimension(1:3, 1:N) :: X,M,Grav
! calculates acceleration on body i due to body j in entries (:,i,j)
         real,array (1:3, 1:N, 1:N) :: A, Xi, Xj, Ms, D, R
         logical,array (1:3, 1:N, 1:N) :: diag
         integer i,j,k

! true for diagonal of N by N slices
         forall (k=1:3, i=1:N, j=1:N)diag = (i. eq. j)

! set up arrays of particles Xi and particles Xj
         Xi = spread (X, dim = 3, ncopies = N)
         Xj = spread (X, dim = 2, ncopies = N)
         Ms = spread (M, dim = 2, ncopies = N)

! displacements and Euclidean distance
         D = Xj - Xi
         R = sqrt (spread (sum (D*D, dim=1), dim=1, ncopies=3))
! calculate accelerations for all pairs except on main diagonal
         where (diag)
          A=0.0
         elsewhere
          A=G*D/R**3
         end where

         Grav=sum (A, dim=3)
end function Grav
```

When this program is run on a parallel machine, the N particles are divided among N_{proc} processors.



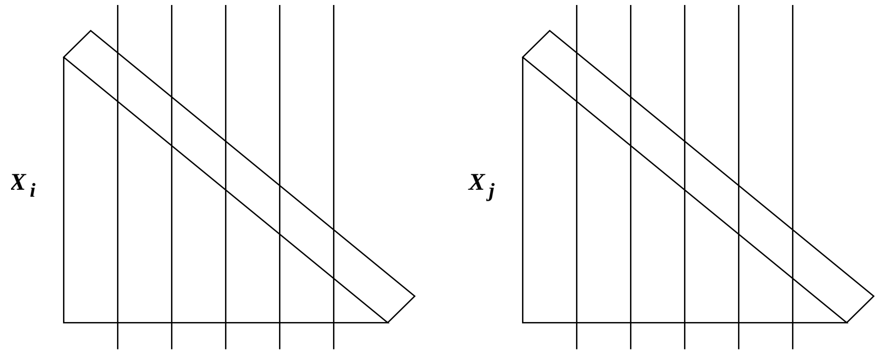
The obvious way to decompose the N^2 particle arrays (and masses) is to divide them into blocks of columns.



Spreading the particles to array X_i only involves copying data within one processor. However, spreading the particles and masses to X_j and M_s requires communication—roughly N_{proc} array elements. Finally, the advantage of the column distribution is that the summation of individual accelerations to get a total for each particle will take place within each

*** missing line ***

For reasonable numbers of particles, the dominant time of the computation is the N^2/N_{proc} floating point computations. The problem with this simple algorithm is that it ignores the symmetry of the forces, i.e., that the force on particle j due to particle i has the same magnitude as the force on particle i due to particle j and differs by sign. We can reduce the floating point computation by a factor of two, if we only calculate non-redundant forces. But this means that our parallel computation is not easily load-balanced.



We can, of course, achieve load balancing with a non-uniform distribution. However, we can achieve our goal of having each processor only performa $N/2 * N_{\text{proc}}$ floating point computations in the elegant pipeline algorithm which follows.

*** missing line ***

benefit that it only makes one copy of the particle information instead of n copies, so that it doesn't use N^2 amount of space. The latter can be a serious restriction of the simple algorithm for the number of particles.

Pipeline (ring) techniques for the N-body problem

The problem is that for each particle i, we must pair its position information with the position information of particle j. For the pipeline algorithm, we set up the computation by making one copy of the position array, the array of masses, and an array to hold the accumulating accelerations. Since the masses must be multiplied by each coordinate of the difference of positions, we set it up by spreading the N masses into a $3 \times N$ array.

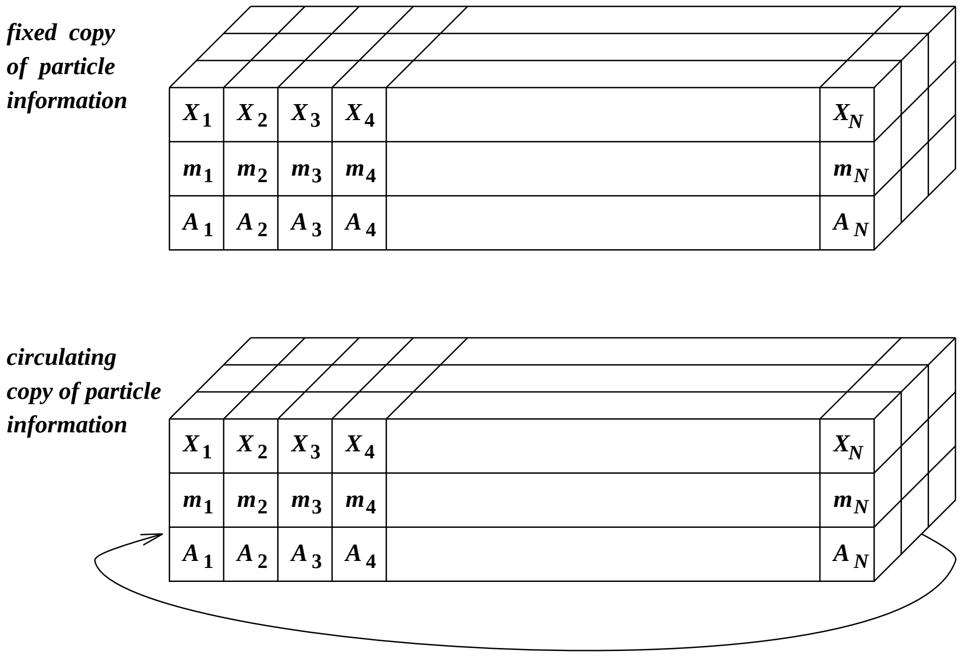


Figure 9: Fixed and cirulcating copies of particle information to set up pipeline.

The basic step of pipeline operation is then to circulate the pipeline copy one step to the right and to calculate accelerations in both directions between the particles that are lined up by index position. Of course, the acceleration between the particles in one direction is just the negation of the acceleration in the other.

Since the accelerations between particles are being calculated in both directions during the operation of the pipeline, it only needs to operate for approximately half of the length of the circulating arrays. More precisely, if N is odd, $(N-1)/2$ steps, each of which calculates $2N$ accelerations, are needed to calculate all $N(N-1)$ accelerations. If N is even, $N/2$ steps are used where all the steps except the last calculate $2N$ accelerations and the last step only calculates the accelerations in one direction.

This pipeline technique can be incorporated into a function called Grav which takes the positions and masses of the particles and calculates the accelerations. In this program, we have chosen to spread these quantities before doing the arithmetic. Furthermore, the quantities $G * m_j$ do not change and we premultiply the constant G.

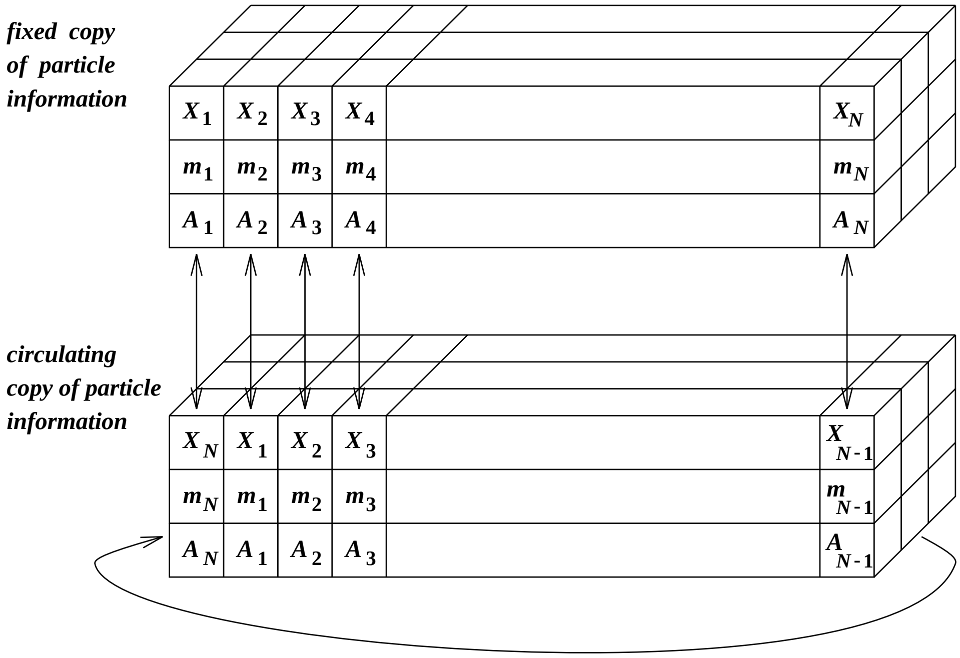


Figure 10: Particle information circulated one step to the right, lining up data for acceleration calculations between particles 1 and N, particles 2 and 1, particles 3 and 2, and so on.

Let P_i be the particle information for particle i:

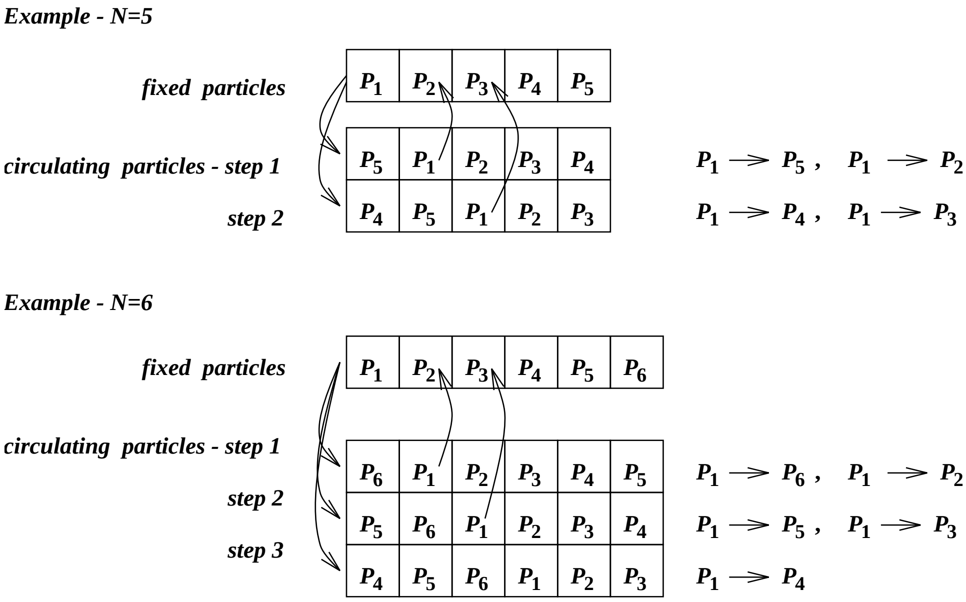
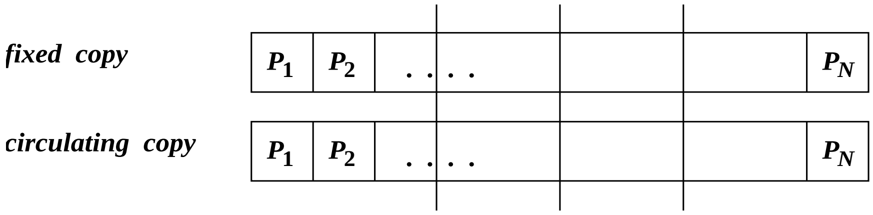


Figure 11: Two small examples showing how the pipeline lines up all particles with particle 1.

```
    function Grav(X,M)
C accepts positions of particles X and masses of particles M
C returns accelerations in Grav
          integer, parameter :: N = number of particles
          real, parameter :: G = gravitational constant
          real, dimension(1:3, 1:N) :: X,M,Grav
          real, dimension(1:3, 1:N) :: A, Xc, Mc, Ac, D, R
          integer k

          A = 0.0          ! fixed accelerations - X and M are used for
                           !   fixed positions and masses
          Ac = 0.0         ! circulating accelerations
          Xc = X           ! circulating positions
          M = G*M          ! precalculate mass * gravitational constant
          Mc = M           ! circulating masses

do k = 1, (N-1)/2
          Xc = cshift (Xc, dim=2, shift= -1)     ! shift circulating arrays
          Mc = cshift (Mc, dim=2, shift= -1)     !    to the right
          Ac = cshift (Ac, dim=2, shift= -1)

          D = Xc - X          ! calculate R to be distance over 3-D coordinates
          R = sqrt (spread (sum (D*D, dim=1), dim=1, ncopies=3))
          D = D/R**3
          A = A + Mc * D          ! fixed acceleration
          Ac = Ac + M * D         ! circulating acceleration
end do

if ((N mod 2) = 0)          ! final one-way accelerations where n is even
          Xc = cshift (Xc, dim=2, shift= -1)
          Mc = cshift (Mc, dim=2, shift= -1)
          Ac = cshift (Ac, dim=2, shift= -1)
          D = Xc − X
          R = sqrt (spread (sum (D*D, dim=1), dim=1, ncopies =3))
          D = D/R**3
          A = A + Mc * D
end if

! combine accelerations for final result-circulating particle in ith
! position corresponds to fixed particle (i - (N-1)/2)
          Grav = A + cshift (Ac, dim=2, shift= (N-1)/2)
    end function Grav
```

In the parallel implementation of the pipeline algorith for *Grav*, both copies of the particle information is decomposed onto processors by columns.



For each processor, we can calculate the running time.

Consider time for one loop iteration of function *Grav*.

- Shifting particles communicates one set of particle information—all processors communicate at the same time (could be 7, but overshadowed by latency):

$$g * t_{\text{comm}}$$

- Floating point calculations: roughly 3N of $-$, *, sum, sqrt, exp, /, *, +, *, +

$$> 30N * t_{\text{float}}$$

The loop is executed for all the particles in the local partition of that processor, plus one more shift, giving a total time for *Grav*:

$$\frac{N}{2N_{\text{proc}}}\left(\left(30\frac{N}{N_{\text{proc}}} * t_{\text{float}}\right) + 9 * t_{\text{comm}}\right)$$

Then the total time for the Runge-Kutta solver is:

$$4 * n_{\text{iter}}\left(\frac{N}{2N_{\text{proc}}}\left(\left(30\frac{N}{N_{\text{proc}}} * t_{\text{float}}\right) + 9 * t_{\text{comm}}\right)\right)$$

Giving $O(N^2)$ running time in the number of particles.

Running the N-body program

For test purposes, we ran our N-body solver on a subset of the solar system consisting of the sun, the planets, and as many "fake" asteroids as required to make an appropriate number of particles in the system. The main program is outlined:

```
program nbody

read input data:
        total number of bodies N
        positions, velocities, and masses of the sun and planets
                X, V, and m
        mass and radius of asteroid group
        total time interval t
        number of steps ns
make up a circle of asteroids X, V, and m
spread masses into 3 x N array M
h=t/ns

display solar system at initial position
call runge-kutta (X,V,M,h,ns)
display solar system at time t
end program
```

We ran our trial program on a DECmpp with 8k processors and on a CM5 with 32 nodes (128 vector units) for 100 time steps representing one year in the solar system. A problem size of $N = 64$ bodies is the first size big enough to take advantage of these machines in the N^2 programs. In this case, the $N \times N \times 3$ arrays are of size 12,228. These map onto the DECmpp using 4096 processors and a VP ratio of three, and onto the CM5 with each vector unit having three vectors of length 32.

Times in number of milliseconds per time step:

	DECmpp: N^2	Pipeline	CM5: N^2	Pipeline
$N = 64$	11.1	60		
$N = 256$				
$N = 512$				

Although the pipeline performance is improving, it probably won't really outperform the N^2 algorithm until a problem size of a few thousand particles. This problem size will most likely perform well on MIMD machines of around 16–32 processors as well.

Of course, other algorithms are possible, especially if they take into account the machine architecture. For example, the particles could be grouped by the number that would fit onto one processor. These particles would use either algorithm locally and would also be used as elements in a large array that is pipelined over the processors.

5.4 Real problems

Since there are no dissipative forces which would convert mechanical energy to heat in N-body problems, the motion of the particles, if calculated exactly, would conserve energy (kinetic and potential), linear momentum, angular momentum. It would also have the property that the center of mass of the system has a constant velocity. Fortunately, it is easy to show that any of the numerical methods also obey these conservation laws. But in practical situations, one way to achieve more accurate results is to compute the various energies and momenta at regular intervals and use the conservation laws to either check or help correct the calculation.

The laws of conservation of energy and momentum are incorporated in one numerical method designed explicitly for N-body problems—the Greenspan mechanics. References are noted below.

Although the data parallel techniques discussed here are successful on astrophysical problems of size of a few thousand particles, many problems involve much larger numbers of particles where the accelerations between individual particles cannot be practically computed. Astrophysical problems on the galactic scale fall into this category. In these problems, the system is no longer represented as an unordered set of particles. Instead, the particles are organized by their position in space in a data structure called a Barnes-Hut tree. This is a oct-tree representation of three-dimensional space, in which each internal node of the tree represents an area of space and each of its children represents one-eighth of that space until the leaf nodes represent one particle. For each internal node, an approximation of the force of all the particles in that space is calculated. Then for each particle, instead of calculating all $N-1$ forces due to other particles, the approximations are used to calculate the force due to all the particles which are farther away than some distance. Of course, as the system evolves over time, the motion of particles requires that the Barnes-Hut tree be dynamically changed as particles move from one part of space to another. Such structures are not naturally represented in the static arrays that are currently used for synchronous data parallel computing and are the subject of ongoing research. More details on solving N-body problems using Barnes-Hut trees can be found in [Salmon].

5.5 References

N-body problems

Marciniak, A., *Numerical Solutions of the N-body Problem*, D. Reidel Publishing Co., Dordrecht, 1985.

Greenspan, D., *Arithmetic Applied Mathematics*, Pergamon Press (International Series in Nonlinear Mathematics), Oxford, 1980.

LaBudde, R. A., and Greenspan, D., "Energy and momemtum conserving methods of arbitrary order for the numerical integration of equations of motion," *Numerische Mathematik*, **26**, 1–16, 1976.

Salmon, John K., *Parallel Hierarchical N-Body Methods*, Dissertation from Caltech. Technical Report SCCS-52, CRPC-90-14, 1990.

Ordinary Differential Equations

Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T., *Numerical Recipes: The Art of Scientific Computing*, Second edition, Cambridge University Press, 1992.

Henrici, P., *Discrete Variable Methods in Ordinary Differential Equations*, John Wiley & Sons, Inc., New York, 1962.

Burden, Richard L. and Faires, J. Douglas, *Numerical Analysis*, Fourth edition, PWS-Kent Publishing Company, 1989.