Short Course on Parallel Computing – Exercises

Peter Bastian, Olaf Ippisch, Jorrit Fahlke, Christian Engwer, Dan Popović

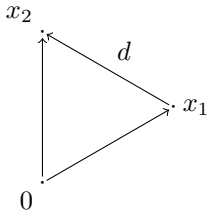
Exercise 1 Gravitational N-Body Problem (GNBP)

Make yourself familiar with the program `nbody_vanilla.c`. It computes the movement of a number of bodies in empty space acting on each other through gravitation. Read the description below. Read the code, compile the code, run the code, look at the output files with `paraview`. Twiddle with the code (for instance by using different initial conditions) and observe how the output changes. Look at the MFLOPS rate and note how it changes (or does not change) with the number of bodies.

For a more in-depth treatment of the problem see "The Art of Computational Science – How to build a computational lab" (`www.artcompsci.org`).

1 The Problem

Consider two bodies floating in empty space:



$$x_1 + d = x_2$$
$$\Rightarrow d = x_2 - x_1$$

This is the 2-body case. Newton's law gives us an expression for the force acting on body 1 due to body 2:

$$F_{12}(t) = \gamma \frac{m_1 m_2}{\|x_2 - x_1\|^3}(x_2 - x_1) \tag{1}$$

With more than two bodies (N-body case) the total force acting on body i is the sum of all the forces exerted by the other bodies:

$$F_i(t) = \sum_{\substack{j=0 \\ j\neq i}}^{N-1} \gamma \frac{m_i m_j}{\|x_j - x_i\|^3}(x_j - x_i) \qquad \forall i = 0, \ldots, N-1 \tag{2}$$

Newton's second law relates the acceleration of one body with the force acting on it

$$F_i(t) = m_i a_i(t) \qquad \forall i = 0, \ldots, N-1. \tag{3}$$

With the definition of the acceleration

$$a_i(t) = \frac{dv_i(t)}{dt}, \qquad v_i(t) = \frac{dx_i(t)}{dt} \qquad \forall i = 0, \ldots, N-1 \tag{4}$$

we arrive at the following system of linear ordinary differential equations (ODEs):

$$\left.\begin{aligned} \frac{dx_i(t)}{dt} &= v_i(t) \\ \frac{dv_i(t)}{dt} &= \sum_{\substack{j=0 \\ j\neq i}}^{N-1} \gamma \frac{m_j}{\|x_j - x_i\|^3}(x_j - x_i) \end{aligned}\right\} \qquad \forall i = 0, \ldots, N-1 \tag{5}$$

In the following we place the unknowns $x_i(t)$ in a vector $x(t)$ in the following way:

$$x(t) = \begin{pmatrix} x_0(t) \\ \vdots \\ x_{N-1}(t) \end{pmatrix} = \begin{pmatrix} (x_0(t))_0 \\ (x_0(t))_1 \\ (x_0(t))_2 \\ (x_1(t))_0 \\ \vdots \end{pmatrix} \tag{6}$$

Similar for a, v and m.

2 Centre-of-Gravity Correction

The position of the centre of gravity and its velocity are

$$R(t) = \frac{1}{M}\sum_{i=0}^{N-1} m_i x_i \qquad V(t) = \frac{1}{M}\sum_{i=0}^{N-1} m_i v_i \qquad \text{where } M = \sum_{i=0}^{N-1} m_i. \tag{7}$$

The velocity $V(t)$ is constant, since there are no external forces acting on the system, and the centre of gravity moves in a straight line. Thus, without loss of generality, we can fix the centre of gravity at the origin:

$$\left.\begin{aligned} \tilde{x}_i(0) &= x_i(0) - R(0) \\ \tilde{v}_i(0) &= v_i(0) - V(0) \end{aligned}\right\} \qquad \forall i = 0, \ldots, N-1 \tag{8}$$

This correction is done once at the beginning. The $\tilde{}$ is omitted in the following and it is assumed that $R(t) = V(t) = 0$.

3 Softening

When two bodies collide $x_j \to x_i$ the force exerted by one particle on the other becomes infinite: $F_{ij} \to \infty$. Especially in computing infinity is highly undesired. To avoid this, we modify the denominator in the acceleration such that it never becomes never becomes zero:

$$a_{ij} = \frac{\gamma m_i}{(\|x_j - x_i\|^2 + \epsilon^2)^{3/2}}(r_j - r_i) \tag{9}$$

The corresponding force may be derived from the potential

$$\Phi(x_i, x_j, \epsilon) = -\frac{\gamma m_i m_j}{(\|x_j - x_i\|^2 + \epsilon^2)^{1/2}}. \tag{10}$$

The physical interpretation is that the original force equation holds for a point-like body, while the new equation holds for a body with non-zero spatial extend and a variable density distribution of

$$\rho(r) \propto \frac{1}{(r^2 + \epsilon^2)^{5/2}}. \tag{11}$$

4 Numerics

A single-step discretization scheme For the time discretization a single step scheme with the following iteration rule is used:

$$x^{k+1} = x^k + v^k \cdot \Delta t + a^k \frac{(\Delta t)^2}{2} \tag{12}$$

$$v^{k+1} = v^k + (a^k + a^{k+1}) \cdot \frac{\Delta t}{2} \tag{13}$$

This scheme has the advantages, that

- it requires only one evaluation of a per step (as long as a^k and a^{k+1} are stored),
- Δt may vary per step (energy conservation is lost in that case, however), and

- it is a second order scheme.

The second order accuracy can be shown by doing the Taylor expansion of $x(t)$ and $v(t)$. For $x(t)$ we have:

$$x(t+\Delta t) = x(t) + \Delta t \frac{dx(t)}{dt} + \frac{\Delta t^2}{2}\frac{d^2v(t)}{dt^2} + O(\Delta t^3) \tag{14}$$

$$= x(t) + \Delta t v(t) + \frac{\Delta t^2}{2} a(t) + O(\Delta t^3) \tag{15}$$

$$\rightsquigarrow x^{k+1} = x^k + v^k \cdot \Delta t + a^k \frac{(\Delta t)^2}{2} \tag{16}$$

And similarly for $v(t)$:

$$v(t+\Delta t) = v(t) + \Delta t \frac{dv(t)}{dt} + \frac{\Delta t^2}{2}\frac{d^2v(t)}{dt^2} + O(\Delta t^3) \tag{17}$$

$$= v(t) + \Delta t a(t) + \frac{\Delta t^2}{2}\frac{da(t)}{dt} + O(\Delta t^3) \tag{18}$$

The remaining time derivative $\frac{da(t)}{dt}$ can be handled by the Taylor expansion of $a(t)$:

$$\frac{da(t)}{dt} = \frac{a(t+\Delta t) - a(t)}{\Delta t} + O(\Delta t), \tag{19}$$

which leads to

$$v(t+\Delta t) = v(t) + \frac{\Delta t}{2} a(t) + \frac{\Delta t}{2} a(t+\Delta t) + O(\Delta t^3) \tag{20}$$

$$\rightsquigarrow v^{k+1} = v^k \frac{\Delta t}{2} a^k + \frac{\Delta t}{2} a^{k+1} \tag{21}$$

Thus for local truncation error holds the order of consistency of 2.

5 Implementation

Structure of the Data We have to store x^k, v^k, a^k, a^{k+1} and m. There are the following options to store these

1. field of `struct { double x[3], v[3], a[3], anew[3], m; }`,
2. `double x[N][3], v[N][3], a[N][3], anew[N][3], m[N]`,
3. `double x[3][N], v[3][N], a[3][N], anew[3][N], m[N]`.

The current implementation uses option 2.

Initial Conditions To generate initial conditions, there are two functions in `generate_vanilla.h`:

`plummer()` All bodies have equal mass $\frac{1}{N}$, corresponding to a macroscopic density leading to the "Plummer potential"

$$\Phi(r) = -GM \frac{1}{(r^2 + a^2)^{1/2}}. \tag{22}$$

and velocity accordingly. The system is in equilibrium. Centre-of-mass correction is applied. For more on the Plummer model see `http://www.artcompsci.org/kali/vol/plummer/title.html`

`cube()` All bodies randomly in a cube with zero velocity. Mass is random in $m \pm \Delta m$. Centre-of-mass correction is applied.

File I/O Data is stored in "VTK" format (Visualisation Toolkit, `www.vtk.org`). These files can be examined with `paraview` (`www.paraview.org`). The functions to read and write such files are `read_vtk_file_double()` and `write_vtk_file_double()` from `io_vanilla.h`. This can be used to restart if the program has to be aborted, or to use pregenerated initial condition.

Time Measurement The function `double get_time()` from `stopwatch.h` returns some time in seconds. "Some time" means that the "time" that `get_time()` measures has $t = 0$ at some unspecified point time, so it only makes sense to use differences of the values returned by `get_time()`. It *does* measure the "wall time" and not the CPU time of the process: the CPU time of a process is the sum of the time slices that the operating system has given to a thread of that process. So a process with two threads on a two-CPU system can use two seconds of CPU time while the wall time advances only by one second.

Main Program

`main()` The `main()` function reads the program arguments, allocates the variables and initialises them to the initial conditions. It then does the loop over the time steps, calling `leapfrog()` for each step and writing the output file and some statistics every `mod` time steps.

`leapfrog()` This function does exactly one time step. The name "leapfrog" is a little inaccurate: There is a time discretization by the name "Leapfrog", and the one step scheme is related to it, but it is not exactly the same.

Input parameters: the number of bodies `n`, the time step size `dt` and the vectors of old positions `x`, of old velocities `v`, of masses `m`, and of old accelerations `a`.

Output parameters: the vectors of new positions `x`, of new velocities `v` and of new accelerations `a`.

There is an additional parameter `aold` which is used by the `leapfrog()` as temporary storage for the old acceleration vectors, but its values on input and output should not be considered meaningful.

`acceleration()` This function is used by `main()` to calculate the initial values for the acceleration from the initial values for the positions. Similarly used by `leapfrog()` to calculate the new vector of accelerations after the new vector of positions has been computed.

Input parameters: the vectors of positions `x` and of masses `m`.

Output parameters: the vector of accelerations `a`. This functions accumulates the acceleration in this vector, which means the vector must be zero-initialised on entry to this function.

The effect of body i on the acceleration of body j is exactly the opposite of the effect of body j on the acceleration of body i: $a_{ij} = -a_{ji}$. The vanilla implementation of the `acceleration()` function exploits this symmetry to save halve the computational effort.

Exercise 2 GNBP with OpenMP

Parallelise `nbody_openmp.c` with OpenMP by inserting the appropriate `#pragma omp` lines for loop parallelisation and critical sections. `nbody_openmp.c` is a copy of `nbody_vanilla.c` with some added code to read the number of threads from the command line. Don't bother with the parallelisation of anything but `acceleration()`, since that needs a number of operations proportional to N^2 while everything else needs number of operations proportional to N.

How does the parallel program perform in comparison to the sequential program, for different numbers of bodies?

Hint: Since the parallel and the sequential versions of the program do the computation in a different order, they may produce slightly different results because of the finite precision of `double`s. So you cannot directly compare a `.vtk` file produced by the sequential program with one produced by the parallel program. There is however a script `fuzzy_diff` that can compare to files ignoring differences in numerical values below a certain threshold ϵ. It can be used like this:

```
./fuzzy_diff sequential.vtk parallel.vtk 1e-10
```

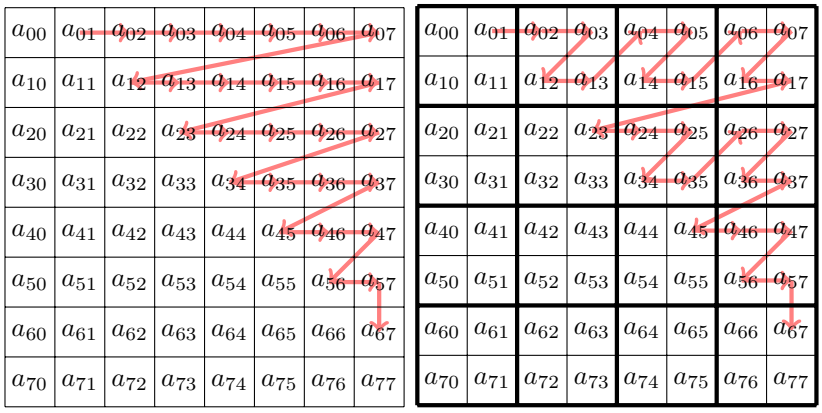
This will compare `sequential.vtk` with `parallel.vtk`, ignoring any differences in numerical values smaller than $\epsilon = 10^{-10}$.

Exercise 3 GNBP with tiling

`nbody_vanilla.c` does not use the CPU cache efficiently. Each time the `acceleration()` function computes the acceleration of body i, it has to iterate over all the bodies $j \in [i+1, \ldots, N-1]$ and load their positions

`x[j]` and masses `m[j]`, compute the acceleration $a_{ij} = -a_{ji}$ and accumulate it in `a[i]` and `a[j]`. If the CPU cache cannot hold all that data, spatial locality is lost and the cache becomes ineffective. This depends very much on the size of the problem: for small enough N you should have observed an increase in the MFLOP rate with `nbody_vanilla.c`.

To make better use of the cache even for bigger problems one can employ a technique known as "tiling". Below, the left figure shows the order in which `nbody_vanilla.c` computes the accelerations. The figure to the right shows the order in which accelerations are computed when tiling is done. This example uses a tile size of $B = 2$.



If the cache is big enough to hold masses for B bodies and the positions and accelerations for $2B$ bodies, then the cache will not overflow while the program is working on one tile. Thus, when working on the second line of a tile the mass and position information will already be in the cache and there is no need to refetch it from main memory.

`nbody_tiled.c` is a copy of `nbody_vanilla.c`. Change the `acceleration()` function to make better use of the cache by tiling. Split the loop over `i` and `j` into outer loops which loop over the tiles and inner loops which loop within the tile, like this:

```
for(I = 0; I < N; I += B)
  for(J = I; J < N; J += B)
    for(i = I; i < MIN(N, I+B); ++i)
      for(j = MAX(i+1,J); j < MIN(N, J+B); ++j) {
        /* code goes here */
      }
```

Can you gain performance over `nbody_vanilla.c`? Is there an optimal tile size?

If you cannot gain performance for any value of N and B up to $2^{14} = 16384$, one possible problem is that the CPU cache is not fully associative. This can lead to aliasing: For a 2-way associative cache and one particle `i` the position `x[i]`, the mass `m[i]` and the acceleration `a[i]` are all loaded into the same set of cache lines. Since for a 2-way associative cache there are two cache lines per set, the different variables force each other out of the cache in turn.

One way around this is to make local copies of the data for each tile, work on that, and add the acceleration back to the global data.

Exercise 4 GNBP with tiling and OpenMP

Please write a version of the program that combines tiling with OpenMP. You can start from `nbody_tiled_openmp.c`, which is a copy of `nbody_tiled.c` from the sample solution.