Computer Architecture and Operating Systems Department (CAOS/DACSO)

Engineering School

Universitat Autònoma de Barcelona

LAB: Parallel Programming

C Programming and Performance Engineering

Lab Assignment

C Programming and Performance Engineering

MAIN GOAL:

C Programming for CPU and Performance Engineering using one of the two applications that we describe in this document (the code for the baseline version is available in the ../alumnos directory, in your account).

SPECIFIC GOALS:

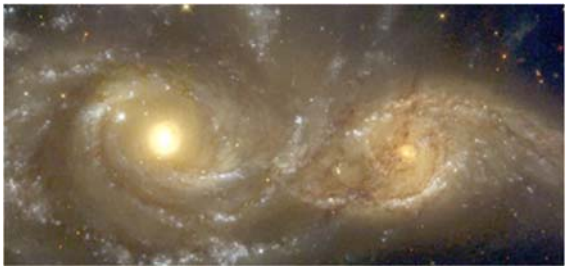
- Understand the algorithm and its computational model (complexity, problem size)
- Performance engineer the code: optimize the serial execution and understand where is the performance bottleneck
- Identify opportunities of vectorization (use of SIMD instructions) and improvement of the memory accesses

Detailed Schedule

1. Select one of the applications described in this document. Define the operation related to the application that will be used as a unit to measure the performance (operations/second). Estimate the algorithmic complexity of the application: i.e., find a formula to compute the total number of operations performed as a function of the problem size (one or more parameters related to the input of the application).
2. Compile the base version of the program and measure relevant performance metrics (time, instructions executed, IPC). Select a problem size that is big enough for the execution to take several seconds. Determine a way to check that the output of the program is correct: remember that the discrete nature of the mathematical operations using real numbers (floating-point) implemented in the computer means that the associative property does not hold for additions and multiplies. Therefore, changes in the order of the operations may generate slight variations in the results that can grow if the accumulated error diverges.
3. Measure execution time to check the effect of the problem size on performance. Identify the time taken by the initialization stages and the time taken by the main part of the algorithm. Ideally, the initialization part should be relatively less and less important as the problem size grows. By selecting the data type used for real numbers, either float or double, you can test the effect both on performance and functionality.
4. You can check the performance differences of the code generated by different compilers (different versions of the gnu gcc or intel icc) and using different compilation flags or options.
5. Improve the program implementation of the serial (or single-thread) baseline code in order to achieve better performance. There are several classical optimizations that you should consider: loop interchange (or reordering), code motion, strength reduction, … Ask for help to your teacher. Explain your results, both when the optimizations succeed and when they fail.
6. Improve the program implementation of the serial code in order to take advantage of the SIMD or vector instructions available in the processor. Ask the teacher if you need help.
7. Find out the performance bottleneck of the program (using `perf`). Ask the teacher if you need help.

Deliver a document with an explanation of your work and your results, with a few but well-designed figures, on the corresponding task in the Campus Virtual, before the deadline date. Identify the bibliographical sources of your work: it is common sense and advisable that you get information from internet, but you should refer to your sources. You can optionally work on both applications, but quality is preferable to quantity.

N-body simulation using all pairwise forces



The classical *N-body* problem simulates the evolution of a system of *N* bodies in a tridimensional space, where the force exerted on each body arises due to its interaction with all the other bodies in the system. *N*-body algorithms have numerous applications in areas such as astrophysics, molecular dynamics and plasma physics. The simulation proceeds over time-steps, each time computing the net force on every body and thereby updating its position and other attributes. If all pairwise forces are computed directly, this requires $O(N^2)$ operations at each time-step. Hierarchical tree-based methods have been developed to reduce the complexity. We will focus only on the algorithm using all the pairwise forces and assuming that the only force is gravity. The next figure shows the equations that define the acceleration of a body Ri provoked by all the other bodies Rj.

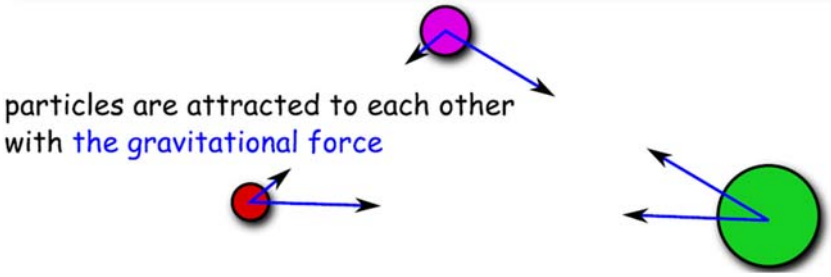
Gravitational N-body dynamics:

Newton's law of universal gravitation:

$$M_i \vec{R}_i''(t) = G \sum_j \frac{M_i M_j}{\left|\vec{R}_i - \vec{R}_j\right|^3} \left(\vec{R}_j - \vec{R}_i\right)$$

where:

$$\left|\vec{R}_i - \vec{R}_j\right| = \sqrt{(R_{i,x} - R_{j,x})^2 + (R_{i,y} - R_{j,y})^2 + (R_{i,z} - R_{j,z})^2}$$



As shown by the next code, the program iterates over a predefined number of time slots for the final position and velocity of the N bodies. In each time iteration the current position of all bodies (at time t) is used to generate the aggregated force exerted on each body, and then this force and the velocity of the body are used to compute the new velocity and position of each body (at time t + Δt). The function that computes the pairwise force exerted between two bodies is presented below. The distance of the two bodies is computed by adding the squares of the distances in each space dimension, and then computing the square root. Before computing the square root, a small constant SOFTENING_SQUARED is added in order to reduce the effect of the computation error for bodies that are very close. A side effect of this softening method is that the force computed between a body and itself is correctly computed as 0.0, and then there is no need to handle this case using special code.

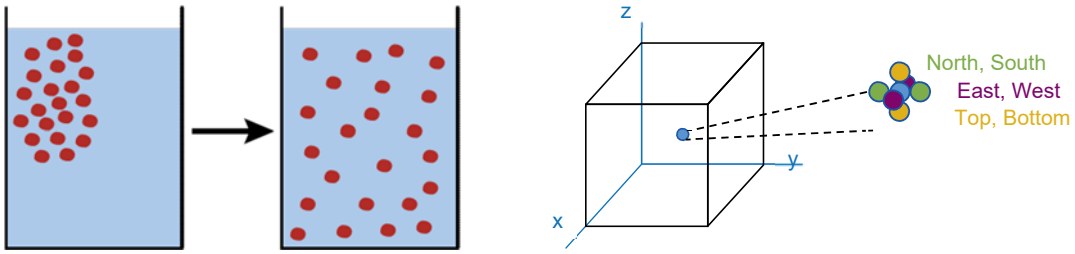
```
for t=0 to time    // TIME
    for i=0 to n       // for each body i
        for j=0 to n       // for each body j, add force
            force[i] += bodyBodyInteraction(pos[i], pos[j]);
    for i=0 to n    // for each body i
      pos[i], vel[i] = advance_step( pos[i], vel[i], force[i], dt)

bodyBodyInteraction:
  real rx, ry, rz = jPosx – iPosx, jPosy – iPosy, jPosz - iPosz;
  real distSqr = rx*rx+ry*ry+rz*rz;
  distSqr += SOFTENING_SQUARED;
  real s = jMass / POW(distSqr,3.0/2.0);
  result is { rx * s, ry * s, rz * s};
```

Performance tests should be made with N=20000 and time= 100, and both with real numbers of type float and double. The constant FP32 must be defined at compile time to use float numbers (example: gcc –DFP32=1 –O3 …).

Diffusion Algorithm (3D stencil)

The program is part of a real scientific application developed by Naoya Maruyama from the Riken Advanced Institute for Computational Science in Japan. The purpose of the program is to simulate diffusion of a solution over time and through a volume of liquid in a three-dimensional region as shown in the figure below. The total volume is divided into smaller pieces of cubic shape and constant size, and each cube is given a concentration of the solution.



The program iterates over a predefined number of time slots for the final state of the solution. In each time iteration, the current status of all portions of the liquid region (at time t) is used to generate the new state (at time t + Δt). Six neighboring sub-volumes (east, west, north, south, top and bottom) are used to obtain the next state of each cube. Therefore, each of the three dimensions of the volume is iterated in each time iteration, applying a function at each point that multiplies the density at each point by a specific weight. The operation resulting in the new value in each point is called a *stencil* (a 3-dimensional *seven-point stencil* in this case), and it is used to calculate the joint effect of the six neighboring sub-volumes over time.

Let F^t be the 3-dimensional array that contains the state of concentration at each point in time t. The stencil for an interior point (x, y, z) is defined as follows, where constants Cx should be understood by replacing x by a letter with the following interpretation (C: center, W: west, E: east, N: north, S: south, B: bottom, T: top):

$$F^{t+1}_{x,y,z} = CC \times F^{t}_{x,y,z} + CW \times F^{t}_{x-1,y,z} + CE \times F^{t}_{x+1,y,z} + CN \times F^{t}_{x,y-1,z} + CS \times F^{t}_{x,y+1,z} + CB \times F^{t}_{x,y,z-1} + CT \times F^{t}_{x,y,z+1}$$

The code uses two 3D matrices, F1[] and F2[], to store data volumes. In one iteration F1[] contains the input data and F2[] is used to store the result, and in the next iteration the arrays swap roles (this algorithmic strategy it is known as *double buffer*). As shown in the code bellow, there are four nested loops, one for iterating on time, and three for iterating on each of the dimensions. The body of the inner loop applies a 7-point stencil using the sub-section volume in the center and its neighbors north, south, east, west, top and bottom. The special cases of the sub-volumes that are on the edge of the 3-dimensional region need to be considered. These *boundary conditions* occur when dimensions x, y or z are zero or when they have the maximum width, height or depth, represented by the variables NX, NY and NZ. In these cases, the value of the neighbor that is outside the volume is replaced by the density value of the central element being calculated. This provides a reasonable approximation to the diffusion in the border region.

```
for t=0 to time              // TIME
  for (x,y,z) in 3D-volume   // SPACE
    F2[x,y,z]= STENCIL_7p ( F1[], x, y, z, Weights[7] )
  swap_pointers(F1,F2)
```

The size of the 3D-region is by default 128x128x128, but you can change that by passing the appropriate arguments at run time. The code initializes the volume of liquid with a random data generated according to a certain distribution. Performance tests should be made with the problem size NX,NY,NZ= 128, which represents a cube of 128 × 128 × 128 elements, and both with real numbers of type float and double (you should redefine the constant REAL defined in the code as float or double, as desired).