Bach Khoa University Of Technology
Faculty of Computer Science & Engineering

Course: Parallel Processing
Lab #10 - NBody problem in Parallel & Speed Up

Minh Thanh Chung

October 29, 2017

Goal: This lab helps student to practice with a famous case study in parallel with OpenMP: Nbody. Then, we can improve programming skill in parallel.

Content: The lab shows the problem, then instruct how to solve it from sequence implementation to parallel approach.

Result: After this Lab, student can understand the case studies that need to consider for optimizing the performance of parallel programs.

Contents

1 Nbody simulation: basic algorithm

The N-body simulation refers to the computational problem of solving the equations of motion of gravitationally or electrostatically interacting particles. These problems are used in astrophysics to model the motion of self-gravitating systems, such as planetary systems, galaxies and cosmological structures, and in molecular physics to model the dynamics of complex molecules and atomic structures.
The general form of particle-particle interaction in N-body problems is given by Equation (1):

$$\vec{F_i} = KC_i \sum_{j \neq i} C_j \frac{\vec{R_j} - \vec{R_i}}{\left|\vec{R_j} - \vec{R_i}\right|^3} \tag{1}$$

Here $\vec{R_i}$ are the position vectors of particles. This equation expresses $\vec{F_i}$, the force on particle i exerted by all other particles. The interaction has inverse-square dependence on distance, i.e., the magnitude of the force exerted by particle j on particle i is inversely proportional to the square of the distance between these particles, $\left|\vec{R_j} - \vec{R_i}\right|$. For the gravitational N-body problem, the coupling K is given by the gravitational constant $G \approx 6.674 \cdot 10^{-11} m^3 kg^{-1} s^{-2}$, and the individual terms C_i and C_j are the mass, in kilograms, for the i-th and j-th particle, respectively. For electrostatical problems K is the Coulomb constant $k_e \approx 8.988 \cdot 10^9 N m^2 C^{-2}$ (in the International System of Units, or SI) and the individual terms C_i and C_j are the i-th and j-th charges in Coulombs, respectively.
We assume that all particle masses or charges have the same value: $C_i = C_j$. It is trivial to extend our analysis to the case when C differs from particle to particle. Without loss of generality, we can assume $K = 1$ and $C_i = 1$. Indeed, it is always possible to choose a system of units in which this is true. In the following expressions, $\vec{R}$, $\vec{V}$ and t are scaled quantities in this special system of units, rather than in SI.
Using the forward Euler method for the integration of differential equations, we can express the components of particle velocity $\vec{v_i}$ at the end of the simulation time step Δt as

$$v_{x,i}(t + \Delta t) = v_{x,i}(t) + \Delta t \sum_{j \neq i} \frac{(x_j - x_j)}{[(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2]^{\frac{3}{2}}} \tag{2}$$

$$v_{y,i}(t + \Delta t) = v_{y,i}(t) + \Delta t \sum_{j \neq i} \frac{(x_j - x_j)}{[(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2]^{\frac{3}{2}}} \tag{3}$$

$$v_{z,i}(t + \Delta t) = v_{z,i}(t) + \Delta t \sum_{j \neq i} \frac{(x_j - x_j)}{[(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2]^{\frac{3}{2}}} \tag{4}$$

and the coordinates at the end of the time step as

$$x_i(t + \Delta t) = x_i(t) + v_{x,i}(t + \Delta t)\Delta t \tag{5}$$

$$y_i(t + \Delta t) = y_i(t) + v_{y,i}(t + \Delta t)\Delta t \tag{6}$$

$$z_i(t + \Delta t) = z_i(t) + v_{z,i}(t + \Delta t)\Delta t \tag{7}$$

The quantities shown here are the particle coordinates and the components of particle velocities, i.e., the position vector $\vec{R_i} \equiv (x_i, y_i, z_i)$ and the velocity vector $\vec{R_i} = \vec{V_i} \equiv (v_{x,i}, v_{y,i}, v_{z,i})$.
Algorithm (2) - (7) may be familiar to some readers from applications other than astrophysics or bio-physics: this algorithm is used as a test workload in some software development kits. Note that this algorithm is not particularly efficient nor accurate enough for practical large-scale simulations. This algorithm has a computational complexity of $O(N^2)$ and employs the forward Euler scheme, which is only first-order accurate in time, and unstable for large time steps. In practical scientific applications, the calculation of gravitational

force on each particle is usually performed with tree algorithms, such as the Barnes-Hut method. Tree algorithms can reduce the computational complexity to $O(NlogN)$. In order to solve the equations of motion, explicit or implicit Runge-Kutta methods may be used instead of the forward Euler scheme. These methods can improve the accuracy to fourth order in time and greatly expand the algorithm stability range.

2 Implementation

The source code of this lab includes 2 files: *Makefile* and *nbody.cc*. This algorithm is implemented based on (2) - (7). The process of compiling and running nbody program is shown in *Makefile*.

```
/* nbody.cc  */
#include <cmath>
#include <cstdio>
#include <mkl_vsl.h>
#include <omp.h>

struct ParticleType {
  float x, y, z;
  float vx, vy, vz;
};

void MoveParticles(const int nParticles, ParticleType* const particle, const float dt) {

  // Loop over particles that experience force
  for (int i = 0; i < nParticles; i++) {

    // Components of the gravity force on particle i
    float Fx = 0, Fy = 0, Fz = 0;

    // Loop over particles that exert force: vectorization expected here
    for (int j = 0; j < nParticles; j++) {

      // Avoid singularity and interaction with self
      const float softening = 1e-20;

      // Newton's law of universal gravity
      const float dx = particle[j].x - particle[i].x;
      const float dy = particle[j].y - particle[i].y;
      const float dz = particle[j].z - particle[i].z;
      const float drSquared  = dx*dx + dy*dy + dz*dz + softening;
      const float drPower32  = pow(drSquared, 3.0/2.0);

      // Calculate the net force
      Fx += dx / drPower32;
      Fy += dy / drPower32;
      Fz += dz / drPower32;

    }

    // Accelerate particles in response to the gravitational force
    particle[i].vx += dt*Fx;
    particle[i].vy += dt*Fy;
    particle[i].vz += dt*Fz;
```

```
  }

  // Move particles according to their velocities
  // O(N) work, so using a serial loop
  for (int i = 0 ; i < nParticles; i++) {
    particle[i].x  += particle[i].vx*dt;
    particle[i].y  += particle[i].vy*dt;
    particle[i].z  += particle[i].vz*dt;
  }
}

int main(const int argc, const char** argv) {

  // Problem size and other parameters
  const int nParticles = (argc > 1 ? atoi(argv[1]) : 16384);
  const int nSteps = 10;  // Duration of test
  const float dt = 0.01f; // Particle propagation time step

  // Particle data stored as an Array of Structures (AoS)
  // this is good object-oriented programming style,
  // but inefficient for the purposes of vectorization
  ParticleType* particle = new ParticleType[nParticles];

  // Initialize random number generator and particles
  VSLStreamStatePtr rnStream;
  vslNewStream( &rnStream, VSL_BRNG_MT19937, 1 );
  vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD,
            rnStream, 6*nParticles, (float*)particle, -1.0f, 1.0f);

  // Perform benchmark
  printf("\n\033[1mNBODY Version 00\033[0m\n");
  printf("\nPropagating %d particles using 1 thread on %s...\n\n",
      nParticles,
#ifndef __MIC__
      "CPU"
#else
      "MIC"
#endif
      );
  double rate = 0, dRate = 0; // Benchmarking data
  const int skipSteps = 3; // Skip first iteration is warm-up on Xeon Phi coprocessor
  printf("\033[1m%5s %10s %10s %8s\033[0m\n", "Step", "Time, s", "Interact/s", "GFLOP/s"); fflush(sto
  for (int step = 1; step <= nSteps; step++) {

    const double tStart = omp_get_wtime(); // Start timing
    MoveParticles(nParticles, particle, dt);
    const double tEnd = omp_get_wtime(); // End timing

    const float HztoInts   = float(nParticles)*float(nParticles-1) ;
    const float HztoGFLOPs = 20.0*1e-9*float(nParticles)*float(nParticles-1);

    if (step > skipSteps) { // Collect statistics
      rate  += HztoGFLOPs/(tEnd - tStart);
```

```
      dRate += HztoGFLOPs*HztoGFLOPs/((tEnd - tStart)*(tEnd-tStart));
    }

    printf("%5d %10.3e %10.3e %8.1f %s\n",
        step, (tEnd-tStart), HztoInts/(tEnd-tStart), HztoGFLOPs/(tEnd-tStart), (step<=skipSteps?"*":'
    fflush(stdout);
  }
  rate/=(double)(nSteps-skipSteps);
  dRate=sqrt(dRate/(double)(nSteps-skipSteps)-rate*rate);
  printf("-----------------------------------------------------\n");
  printf("\033[1m%s %4s \033[42m%10.1f +- %.1f GFLOP/s\033[0m\n",
      "Average performance:", "", rate, dRate);
  printf("-----------------------------------------------------\n");
  printf("* - warm-up, not included in average\n\n");
  delete particle;
}
```

```
/* Makefile */
CXX = icpc
CXXFLAGS=-qopenmp -mkl
CPUFLAGS = $(CXXFLAGS) -xhost
MICFLAGS = $(CXXFLAGS) -mmic
OPTFLAGS = -qopt-report -qopt-report-file=$@.optrpt

CPUOBJECTS = nbody.o
MICOBJECTS = nbody.oMIC

TARGET=app-CPU app-MIC


.SUFFIXES: .o .cc .oMIC

all: $(TARGET) instructions

%-CPU: $(CPUOBJECTS)
     $(info )
     $(info Linking the CPU executable:)
     $(CXX) $(CPUFLAGS) -o $@ $(CPUOBJECTS)

%-MIC: $(MICOBJECTS)
     $(info )
     $(info Linking the MIC executable:)
     $(CXX) $(MICFLAGS) -o $@ $(MICOBJECTS)

.cc.o:
     $(info )
     $(info Compiling a CPU object file:)
     $(CXX) -c $(CPUFLAGS) $(OPTFLAGS) -o "$@" "$<"

.cc.oMIC:
     $(info )
     $(info Compiling a MIC object file:)
     $(CXX) -c $(MICFLAGS) $(OPTFLAGS) -o "$@" "$<"
```

```
instructions:
	$(info )
	$(info TO EXECUTE THE APPLICATION: )
	$(info "make run-cpu" to run the application on the host CPU)
	$(info "make run-mic" to run the application on the coprocessor)
	$(info )

run-cpu: app-CPU
	./app-CPU 65536

run-mic: app-MIC
	scp app-MIC mic0:~/
	ssh mic0 LD_LIBRARY_PATH=$(MIC_LD_LIBRARY_PATH) ./app-MIC 65536

clean:
	rm -f $(CPUOBJECTS) $(MICOBJECTS) $(TARGET) *.optrpt
```

The main workload of this program is function named $MoveParticles()$. The outer for-loop over index i calculates the force acting on each particle and changes the particle momentum in response to that force. The inner for-loop over index j calculates the force acting on particle i. For every set of x, y and z, the calculation of variables dx, dy, dz, $drSquared$ and $drPowerN32$ can be performed independently using the same sequence of operations. We must also point out the data structure chosen for the code. Each particle is represented by an object of derived type $ParticleType$. This type is a structure containing the coordinates of the particle and its velocity components. This kind of data structure is not the best representation for optimization purposes.

Task0: Study the code, then compile and run the application to get the baseline performance. To run the application on the host, use the command "*make run-cpu*" and for coprocessor, use "*make run-mic*".

3 Optimized N-body simulation in the C++ language

Task 1: Parallelize MoveParticles() by using OpenMP. Remember that there are two loops that need to be parallelized. You only need to parallelize the outer-most loop.

- Also modify the print statement in, which is hardwired to print "1 thread" (i.e., print the actual number of threads used).
- Compile and run the application to see if you got an improvement.

Task 2: Apply strength reduction for the calculation of force (the j-loop). You should be able to limit the use of expensive operations to one $sqrtf()$ and one division, with the rest being multiplications. Also make sure to control the precision of constants and functions.

- Compile and run the application to see if you got an improvement.

Task 3: Apply strength reduction for the calculation of force (the j-loop). You should be able to limit the use of expensive operations to one $sqrtf()$ and one division, with the rest being multiplications. Also make sure to control the precision of constants and functions.

- Compile and run the application to see if you got an improvement.

Task 4: In the current implementation the particle data is stored in a Array of Structures(AoS), namely a structure of "*ParticleTypes*"s. Although this is great for readability and abstraction, it is sub-optimal for

performance because the coordinates of consecutive particles are not adjacent. Thus when the positions and the velocities are accessed in the loop and vectorized, the data has a non-unit stride access, which hampers performance. Therefore it is often beneficial to instead implement a Structure of Arrays (SoA) instead, where a single structure holds coordinate arrays.

Implement SoA by replacing "*ParticleType*" with "ParticleSet". Particle set should have 6 arrays of size "n", one for each dimension in the coordinates (x, y, z) and velocities (vx, vy, vz). The i-th element of each array is the coordinate or velocity of the i-th particle. Be sure to also modify the initialization in $main()$, and modify the access to the arrays in "$MoveParticles()$". Compile then run to see if you get a performance improvement.

4 Exercises

Let's analyze this application in terms of arithmetic intensity. Currently, the vectorized inner j-loop iterates through all particles for each i-th element. Since the cache line length and the vector length are the same, arithmetic intensity is simply the number of instructions in the inner-most loop. Not counting the reduction at the bottom, the number of operations per iteration is 20, which is less than the 30 that roofline model calls for.

To fix this, we can use tiling to increase cache re-use. By tiling in "i" or "j" by $Tile = 16$ (we chose 16 because it is the cache line length as well as the vector length), we can increase the number operations to $16 * 20 = 320$. This is more than enough to be in the compute-bound region of the roofline mode.

Although the loop can be tiled in "i" or "j" (if we allow loop swap) it is more beneficial to tile in "i" and therefore vectorize in "i". If we have "j" as the inner-most loop each iteration requires three reductions of the vector register (for Fx, Fy, Fz). This is costly as this not vectorizable. On the other hand, if we vectorize in "i" with $tile = 16$, it does not require reduction. Note though, that you will need to create three buffers of length 16 where you can store Fx, Fy and Fz for the i-th element.

Implement tiling in "i". then compile and run to see the performance.

5 Submission

After you finish the exercise, please compress all of **source codes** and **figure** into: lab10_MSSV.zip. The section for submitting the lab will be opened on Sakai. Deadline: 5:00pm, Tue - 07-11-17.