

Optimization and Parallelization of a Molecular Dynamics Application

In this lab, we will learn how to optimize and parallelize an application for multicore and manycore architectures. The source code provided is an N-body simulation, which is a simulation of many particles that gravitationally or electrostatically interact.

The program keeps track of the position and velocity of each particle in the structure "Particle". The simulation is discretized into timesteps. In each timestep, first, the force on each particle is calculated using a direct all-to-all algorithm ($O(n^2)$ complexity). Next, the velocity of each particle is modified using the explicit Euler method. Finally, the positions of the particles are updated using the explicit Euler method.

N-body simulations are used in astrophysics to model galaxy evolution, colliding galaxies, dark matter distribution in the Universe, and planetary systems. They are also used in simulations of molecular structures. Real astrophysical N-body simulations, targeted to systems with billions of particles, use simplifications to reduce the complexity of the method to $O(n \log n)$. However, our toy model is the basis upon which the more complex models are built and is illustrative of the optimization and parallelization methods used for real applications.

In this lab, you will mostly be modifying the function `MoveParticles()`.

0. Study the code, then compile and run the serial application on the CPU and on KNL to get the baseline performance numbers. Use the provided Makefile to compile the serial application. Read the optimization report that is generated by the compiler.
1. Apply strength reduction to the calculation of force (the j-loop).
Limit the use of expensive operations. Also make sure to control the precision of constants and functions. Compile and run the application to see if you get a performance improvement.
2. Parallelize `MoveParticles()` by using OpenMP. Choose which loop is best to parallelize. Also modify the print statement which is hardwired to print "1 thread" to print the actual number of threads used. Compile and run the application to see if you get a performance improvement. Implement other necessary optimizations to get good vectorization of the inner loop. Compile and run the application to see if you get a performance improvement.
3. In the current implementation the particle data are stored in an 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 vectorized loop, the data have non-unit stride access, which hampers performance.

Therefore, it is often beneficial to instead implement a Structure of Arrays (SoA) in which a single structure holds coordinate arrays.

Implement SoA by replacing "ParticleType" with "ParticleSet". Particle set should have six arrays of size n , one for each dimension in the coordinates (x , y , z) and velocities (v_x , v_y , v_z). The i th element of each array is the coordinate position 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 and run to see if you get a performance improvement.

4. In the current version of the code, the vectorized inner j -loop iterates through all particles for each i th particle, which has inefficient cache usage for large n . To fix this, we can use tiling to increase cache reuse. Although the loop can be tiled in i or j (if we allow loop swap), it is more beneficial to tile and vectorize in i for the following reason. If we have j as the inner-most loop, each iteration requires three reduction of F_x , F_y , and F_z , which is costly as it is not vectorizable. On the other hand, if we vectorize in i , reduction is not required. Note however that you will need to create three buffers of tile length where you can store F_x , F_y , and F_z for the i th particle.

Implement tiling in i . Then compile and run to see if you get a performance improvement.

5. Use MPI to further parallelize the application across multiple nodes. To make this work doable in a short amount of time, keep the entire data set on each process. Each MPI process should execute only its portion of the loop in the `MoveParticle()` function. Try to minimize the amount of communication between nodes. You may find the MPI function `MPI_Allgather()` useful. Compile and run the code to see if you get a performance improvement.

You should turn in a separate version of your code for each step of the optimization process, along with a report describing and explaining the optimization technique applied and the resulting performance improvement for each step.