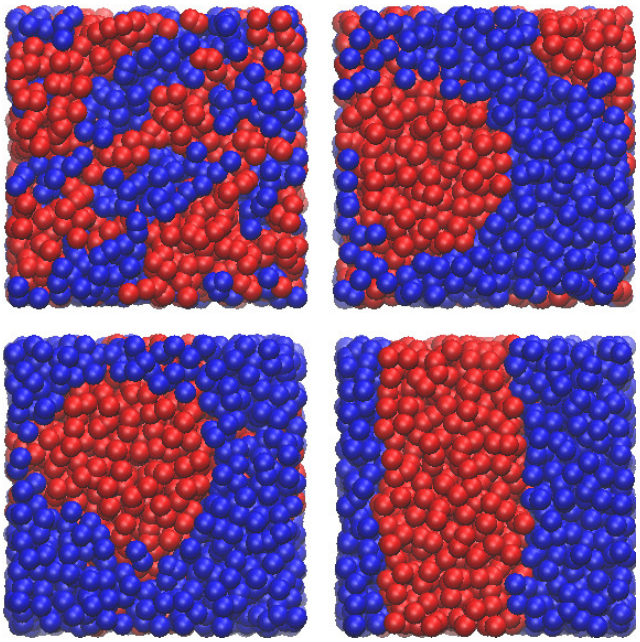
Boosting Dissipative Particle Dynamics



Nursima Çelik & Davide Crisante

In particle simulations, thermostats are used in controlling system temperature, like in real life experiments. In this project, we ported Lowe-Andersen and Peters thermostats to GPU, enabling researchers to use these thermostatting options in reasonable amounts of time.

Conducting scientific experiments on computers gives us a chance to explore more variety of systems in less time. We are able to replicate many real world phenomena on computers and get more insight about mechanisms of nature.

Evidently, it is important to have simulations that produce correct outputs. We don't want our "replication of nature" to deviate from reality and lead us some crazy conclusions. To get correct physics, one thing we need to pay attention is the scale that simulation operates in.

As you may have heard, things work a little different in micro compared to macro scale. While Newton physics is sufficient in defining rules of the world in macro, quantum effects become significant when we go down to the micro level. For this reason, it is good to have different simulation techniques for different scales.

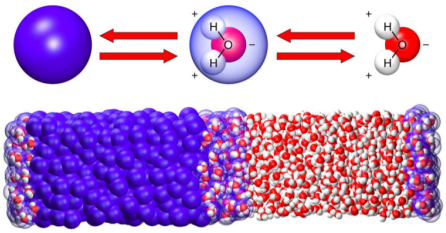


Figure 1: Group of atoms are treated as one particle in DPD.

The scale we are concerned in this project is mesoscale, which is in between micro and macro. Mesoscale can range between 10 –1000 nm and 1 ns –10 ms[1]. One technique used for this scale is called Dissipative Particle Dynamics (DPD).

A glance at DPD technique

In DPD, particles are modelled as spheres, and time is split into small durations called time steps. There may be forces acting on particles which will cause motion. At every time step, we calculate total force applied on each particle, and then we find its next position and velocity. The output is trajectory of every particle as well as system statistics like kinetic/potential energy, temperature, pressure.

In any experiment, it is desirable to keep some quantities (like temperature, pressure, volume) constant to observe the effect of independent variable. Mechanisms for keeping temperature are called thermostats. It is important for simulations to be able to provide thermostats.

Task

There are two versions of DL_MESO code, one written in Fortran language, utilising MPI; the other written in CUDA-C language, utilising GPU. While there are five different types of thermostats in Fortran, only DPD thermostat was available in CUDA. Our task was to port Lowe-Andersen (LA) and Peters thermostats from Fortran to CUDA.

DPD Thermostat vs Lowe-Andersen vs Peters

DPD thermostat is the default option, which is a balance between drag and random forces. Drag forces tend to decrease the temperature by decreasing velocities, and random forces tend to increase it by reproducing the Brownian motion.[citation]

On the other hand, LA and Peters thermostats introduce a velocity correction step after integration instead of drag and random forces. LA and Peters thermostats are similar but yet different. LA applies velocity correction only to a sample of particle pairs instead of all pairs. Also, their equations are different. Specific equations of methods are omitted, but they can be found in the user manual of DL_MESO.

Methodology

We took Fortran code for lowe/peters options as our reference. In every step, it was important to make sure that both versions produced the same results when fed with the same inputs.

We divided code into small steps so that we port one step at a time. Those steps included:

- Velocity Verlet Stage 1
- Force Calculation
- Velocity Verlet Stage 2
- Velocity Correction
- Stress and Kinetical Component Calculation

In order not to get lost during the process, we followed a workflow that looks like as follows.

- Make sure values from both versions matches before the implementation
- Implement the step
- Make sure values from both versions matches after the implementation

We often encountered with mismatched results from serial and parallel versions. Problem was sometimes an uninitialised variable, but sometimes it was not that easy - like one we faced during velocity correction.

Now let's take a closer look to each step.

Step 1: Velocity Verlet Stage 1

Stage velocities of all particles through the half time step.

The default DPD thermostat was already using a kernel called *k_moveParticleVerlet_1* for this purpose. Since lowe/peters had no difference in this step, we used the same kernel in our lowe/peters integration.

Step 2: Force Calculation

Between two Velocity Verlet stages, forces should be calculated.

A particle applies force to all particles around it within a determined distance called cutoff radius. Also, it is affected by only particles those are in the cut-off radius. As effects of particles beyond this radius is considered to be small, their contributions are ignored.

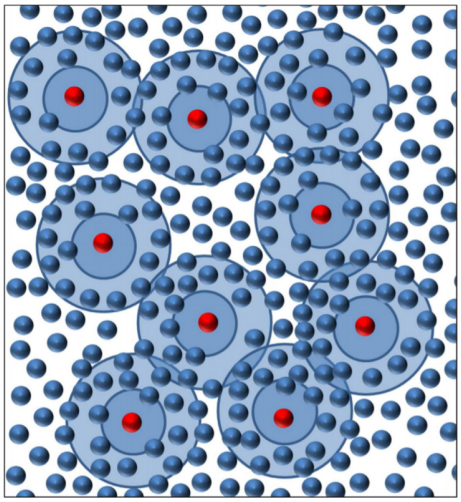


Figure 2: Particles interacting in cut-off radius

In this step, we loop through all particles to find pairs that are interacting with each other. As the result, we get total force applied on each particle at the end.

As you see, force calculation may become a heavy process; especially when the number of particles and chosen step is large. In the results section, we will see that force calculations are indeed take a lot of time. It is the most costly second operation.

To realise this step, we modified a kernel called *k_findForces* that was used for DPD thermostat. We removed random and drag forces.

Step 3: Velocity Verlet Stage 2

Stage velocities up to the end.

This is computationally similar to stage 1. Again, we made use of a kernel called *k_moveParticleVerlet_2* that was used in DPD thermostat. But we needed to make a modification.

Inside this kernel, stress and kinetic energy calculations were done, which was normal for DPD thermostat. However, in lowe and peters, there is a velocity correction step after Velocity Verlet, in which velocities of particles are updated. If we calculated stress or kinetical values in this step, they would no longer be valid after the correction.

So, we created *k_moveParticleVerlet_2_lowe*, removing stress and kinetic energy calculations. We left them to post correction.

Step 4: Velocity corrections

As we mentioned, in DPD thermostat, random and drag forces have the key role to keep temperature around the same level. Here is the innovation of Lowe-Andersen/Peters.

In Lowe-Andersen we needed to

1. Get a random sample of particle pairs
2. Adjust their relative velocities according to "some" equations

For the case of Peters, we needed to

1. Get all particle pairs
2. Adjust their relative velocities according to some "other" equations

As correction was new, there were no kernels we could use as we did for Velocity Verlet. We created a kernel with name *k_correctLowe/k_correctPeters*.

To apply velocity corrections, we needed to get all interacting particles.

In the serial version, the interacting particle pairs were stored in a list while calculating forces. In the correction stage, that list is used to avoid re-traversing all those pairs. The pair list was shuffled. Then in a loop new velocities were calculated and assigned to each pair.

In GPU version, we could make a list of interacting pairs in the force calculation phase, too. But this would cause extra memory transfers, from device to host, and again host to device - something we wanted to avoid.

Therefore, we decided to find pairs once more, this time in order to correct their velocities. Although this is not the most optimal thing to do, it has the advantage of simplicity.

At first, we corrected only a controlled group of particles. That way, we were able to compare the updated velocities from CPU and GPU.

Data Race

In CUDA implementation, data race caused trouble.

If you look to the formulas of lowe/peters once more, you will notice that, we use current velocities of particle i and particle j to get the new ones.

Imagine one thread is on the way to updating pair *(1,2)*, while other wants to update pair *(2,1)*. Finding new velocities depend on the difference between velocities of two particles. When one particle finds correction and updates the velocities of *particle 1* and *particle 2*, the velocity difference in other thread becomes invalid. So, the correction the second thread adds ends up being incorrect.

Incorrect values was a minor problem, because modifications were small. However, the big problem was randomness. Values differed from one execution to other, depending on the order of writes of threads. That made impossible to compare GPU values with those of CPU.

To prevent data race, one option was to use atomic operations. But that would mean to serialise the code. Instead, we made threads to use old velocities instead of current ones. It didn't matter which thread modified velocity first, because nobody read those updated velocities in the correction phase. All threads used the ones before correction.

As a conclusion, we did not strictly obey the formula of thermostats. But, as I mentioned already, this corrections are small enough to allow us to do such a simplification.

Step 5: Stress and Kinetic Energy Calculation

We finally added stress and kinetical component accumulators. This step was important in the sense that we could see if temperature stayed around a determined value as intended.

Results

We tested final program with lowe/peters options both in CPU (32 cores) and GPU. GPU version turned out be 10 times faster.

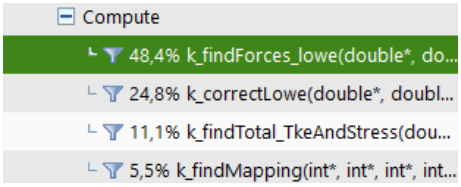


Figure 4: First three most costly functions after memory transfers

We used nvprof program to profile the code. The most costly operation is memory transfers from host to device. Thankfully they happen only once at the beginning of the program. Second and third places come as no surprise, as both force calculation and velocity correction contained a traversal of particle pairs.

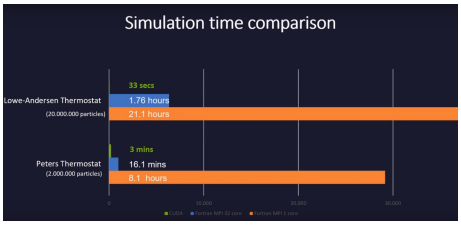


Figure 3: Lowe-Andersen and Peters thermostats execution times

When we compare lowe and peters, we see that lowe takes less time when number of time steps are equal. Remember that velocity corrections were applied to a smaller group of particles in lowe. However, peters has the advantage of allowing one to use larger time steps.

Discussion & Conclusion

We have two new features added to the GPU version of DL_MESO program: Lowe-Andersen and Peters thermostats. We hope that researchers will benefit having these options on GPU. As future work, other thermostats can be ported to CUDA version, like DPD-VV and Stoyanov-Groot. In additon, scaling DL_MESO to large number of GPUs would make a great impact in simulation of larger systems.

References

[1] Michael A. Seaton , Richard L. Anderson , Sebastian Metz William Smith (2013) DL_MESO: highly scalable mesoscale simulations, Molecular Simulation

[2] E. A. Koopman and C. P. Lowe (2006) Advantages of a Lowe-Andersen thermostat in molecular dynamics simulations

PRACE SoHPCProject Title

Scaling the Dissipative Particle Dynamic (DPD) code, DL_MESO, on large multi-GPGPUs architectures

PRACE SoHPCSite

Hartree Centre - STFC, UK

Nursima Çelik

PRACE SoHPCAuthors

Nursima Çelik, Bogazici University, Turkey
Davide Crisante, University of Bologna, Italy

Davide Crisante

PRACE SoHPCMentor

Jony Castagna, Hartree Centre - STFC, UK

PRACE SoHPCContact

Name, Surname, Institution
Phone: +12 324 4445 5556
E-mail: leon.kos@lecad.fs.uni-lj.si

PRACE SoHPCSoftware applied

DL_MESO

PRACE SoHPCMore Information

www.scd.stfc.ac.uk/Pages/DL_MESO.aspx

PRACE SoHPCAcknowledgement

Many thanks to our project mentor Jony Castagna for all the help and patience, and Hartree Centre STFC for the resources.

PRACE SoHPCProject ID

2016