# MATR326 Tools of high performance computing, Final project, topic 3 Parallel 2D molecular dynamics software

Reino Piirilä

March 2024

## Preface

Executing any of the provided shell scripts may require a chmod u+x command on some hardware which may in turn require sudo priviledges. In any case, using the Makefile and ready shell scripts is recommended.

## 1 Problem description

The problem in question is fundamentally an n-body problem. The studied approach in this project is a classical one, which models the bodies as point particles with positions, velocities and an interaction pair potential that describes the potential energy of a pair of interacting bodies. Another thing that the word "classical" implies is something about the rules which all bodies are subject to. In particular, ones that describe how the potential energy of an interaction affects the movement of the bodies participating in that interaction. A way to capture these rules through a set of mathematical expressions is the classical equations of motion formulated in for example Newtonian or Hamiltonian form.
Due to the highly coupled nature of the system of equations, it has a long history of being addressed through numerical means, not the least prominently of which with molecular dynamics. It aims to approximate the phase space state propagation numerically using particular factorization techniques, and essentially calculating its action on each position and velocity coordinate iteratively. In this project, efforts were made to not only program a software capable of such a technique, but to parallelize it to several individual concurrent processes capable of communicating via a distributed memory message passing interface.

# 2 Computer science background

## 2.1 Algorithms

### 2.1.1 Molecular dynamics

Several algorithms were combined in this project of which molecular dynamics is the most central. Its idea is to loop through all the state coordinates of the system and propagate each one of them forward in time by some time step length dt. This is then usually to be repeated over a desired time interval or until some other condition is met.
Unsurprisingly, the delicacies of the method lie in the propagation calculations. For starters, the integrator algorithm itself needs to be carefully established. Secondly, the calculations of each atom's share of interaction potential energies requires one to keep track of the interaction partners of each atom. This is done through so-called neighbor lists, which in this project are unchanging lists of indices of each atom's neighbors. It is assumed that atoms's kinetic energies never reach magnitudes sufficient for bond breaking. In this project a two-dimensional system is studied, so each atom will have four neighbors. The neighbor list contains the indices of the four nearest neighboring atoms of each atom.
Given the anharmonic potential given in the project assignment and how the potential energy of an atom depends on the participants's interatomic distance, the potential energies of each atom must be updated every time the atoms are mapped forward in time with the integrator. And conversely, the updated potential energies then dictate the forces the atoms become subject to, which again gives the atoms acceleration with which another round of propagation may be done.
In this project, periodic boundaries where used, which means that any atom exiting some pre-defined boundary on one axis will enter the cell from the other side on the same axis, and the velocity and neighbors of the atom stay the same. This, in practice, means that any spatial atom position coordinate exceeding a boundary will be subtracted the width of the cell in the physical axis of the coordinate. This means that the simulation cell is practically replicated in space by an infinite amount in each axis.

### 2.1.2 Leapfrog integration

For the time integration scheme itself, the Leapfrog method presented already in the exercise assignment 8 was used, which is a numerical integrator which, for Hamiltonian systems, preserves time-reversibility and the volume of a space element in a flow generated by the Hamiltonian and thus staying true to Liouville's theorem of phase space incompressibility. It is one of several choices of algorithms with these qualities and also numerically a particularly stable one. However, further exploration of the integrator itself will be left out of this project.

## 2.2 Principles of parallelization

### 2.2.1 Domain decomposition

A way to parallelize a simulation of an atomic lattice system is to boldly slice the lattice to several rectangular subdomains and assign them to processes communicating with the processes responsible of the neighboring physical subdomains. Naturally, all but the boundary atoms of a subdomain may be propagated in time without any insight of other subdomains. But the boundary atoms of each subdomain have at least some of their neighbors situated in adjacent subdomains. Because all processes operate on their independent memory banks in a distributed memory message-passing interface, the boundary atom position data must be exchanged between neighboring subdomains on every timestep. Luckily, an open source message-passing standard, MPI, featuring support for C and C++, could be invoked to take care of the division of the system into subdomains among a grid of distributed memory processes shared between a given number of CPU's. The standard also provides functions for taking care of the synchronized sending and receiving of boundary atom position data during the time integration between adjacent processes. In fact, the word adjacent is only made meaningful by MPI's so-called communicators, which create a discrete topology to the grid of processors through the assignment of neighbors. Specifically, MPI_Sendrecv function is one that could be called in each process to synchronize each process sending to a neighbor and waiting to receive from another.
In this project a locally Cartesian, yet globally toroidal, topology was created, which meant assigning exactly four neighboring processes to each process, which are called the northern, southern, eastern and western neighbors, with the twist that this rectangular grid arrangement wraps around its boundaries, going hand-to-hand with the periodic boundary conditions used in the molecular dynamics simulation.

### 2.2.2 Shared memory parallelization

Shared memory parallelization specific to each independent MPI process was also implemented with OpenMP multithreading. It was used to parallelize specific loops with shared memory within each MPI process. This hybrid method was given less emphasis due to its drastically inferior performance in tests compared to plain MPI.

# 3 Documentation

## 3.1 Implementation specifics

The project has two folders, src for the source files and a run folder for object files, executables, python scripts, bash scripts and other outputs. The src folder contains the source files of both the MPI and the serial programs, as well as the hybrid program which combines MPI parallelization and hyperthreading. All of

these source files are written in C++.
The serial program was intentionally made similar to the original fortran program of the exercise assignment. The MPI program begins by setting up the MPI with the cartesian topology communicator with periodic boundary conditions. Then, the master process reads in the input data from the command line and boradcasts it to all other processes, so that the total number of atoms and the number of atoms in one edge of the system could be used, with information about the process's positioning on the process grid, to allocate data arrays for the positions, velocities, accelerations, potential and kinetic energies of all the atoms in a subdomain. Again, since all processes have separate memory, these data vectors are unique to each process. The order of the input parameters is the following: number of unit cells, timestep, simulation time, velocity scaling factor, thermodata output rate and dump data output rate. In addition, the potential factors k2 and k3 can be included after that, but it's not necessary as the program will use default values for them unless specified in the command line.
The atoms of each subdomain are then assigned positions based on the positioning of the process with respect to the Cartesian topology, such that the atoms of two adjacent processes are also each others's neighboring subdomains. This was done with the create_system function. The same function assigns uniform random velocities to the atoms with the Mersenne Twister engine.
To eliminate any random bias in the initial velocities, each process uses get_local_comv to calculate the center of mass velocity of their subdomain, and then calls the subtract_com_velocity which calculates the sum of the center of mass velocities of each subdomain by leveraging MPI_Reduce function in parallel, to calculate the global c.o.m velocity per atom. This is then subtracted from the atoms of each subdomain. The neighbor lists are then also formed before the time integration is begun.
The time integration could then begin. Two custom MPI types were created for the boundary atom position data send and receive buffers, one for data representing atom positions in the top and bottom rows, and another for data representing the atom positions of the left and right columns. Admittedly, I settled for a somewhat unintuitive idiosyncracy when handling the message-passing of each subdomain's boundary data. Since each process needs to receive four different subdomain position matrix edges from four neighboring processes, and to easily avoid having to allocate a separate receive buffer, I simply allocated an extra layer of padding on each edge of a local position array of each subdomain. Specifically, I allocated an empty extra row on top of the top edge, a similar one below the bottom edge, and one left to the left and one right to the right edges. This way the neighbor subdomains's boundary data is, upon reception, gathered straight into the local position data array extra edges, and can be accessed very easily within the same local position data vector with the same logic as all the other atoms in the subdomain. Thus, there's no extra logic needed to find the neighboring positions of the physical edge atoms of a subdomain: the position of the left neighbor of the leftmost physical atom i in a subprocess is still accessed with local_pos[i-1]. The only two caveats are just

that the indexing of the physical atoms starts from num_cols + 4, since the first num_cols + 2 indices belong to the bottom receive buffer, and the (num_cols + 3)th is part of the left receive buffer. The second caveat is that the corner indices, index 0, num_cols + 2 and the upper corner indices are left completely empty, never accessed and thus not part of the receive buffers, since the atoms they would spatially represent are not neighbors of any of the physical atoms of the subdomain. This is illustrated in figure (1). From the drawing one may also get a better look on how the physical atoms are the inner matrix, and how the receive buffers form the padding around it.

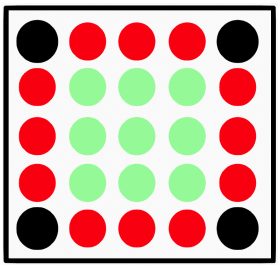

Figure 1: Illustrative drawing of the position array of the atoms of a subdomain. Black circles are the empty corner atoms which are never accessed, rows of red circles represent the receive buffers and green circles the physical atoms of the subdomain.

The actual exchange of boundary atom data was carried out with the MPI_Sendrecv function, which, when supplied with the send and receive buffers and the recipient and source ranks, each process sends a message to their recipient rank and waits to receive the message from their source rank concurrently. This communication was performed in both grid axes, horizontally and vertically. The recipient and source ranks could be obtained with MPI_Cart_shift function since it could be used to find the neighbors of the process with respect to the MPI topology on a given axis when using a stride of 1.

### 3.2 Time integration

Knowing all the positions of the atoms in the subdomain as well as all the positions of their neighbors thanks to the extra receive buffer pads, the accelerations of the atoms were calculated by computing the potential energies based on the interatomic distances with the neighbors with the update_all_accelerations function, which calculates the acceleration of each atom using the accel function. This loop over all atoms, was shared-memory parallelized with OpenMP in the program using the hybrid approach.
Then, finally the positions and velocities may be updated locally in every process

with the leapfrog function which performs leapfrog integration on the coordinates and velocities. This too was parallelized with OMP in the hybrid version.

### 3.3 User's manual

For compilation, a Makefile is provided and its usage recommended as the primary building tool. The Makefile is found in the root process directory and used there. One may compile either the serial or the MPI program with it, or use clean to delete all object files and executables. The Makefile uses -O3 optimization flag by default. The Makefile is in an easily modifiable form, where the user may change only the flag variables at the start of the program and they will apply in all compilation options.
Here's an example use:

```
make clean
make mpi
```

Giving this command in the root process directory creates an mpi executable in the run directory.
Alternatively, for a more manual approach, compilation shell scripts compile_serial.sh and compile_mpi.sh are also included just in case, and for reference on how manual compilation might look. Manual compilation instructions are also included in the README file.
In the run directory, several shell scripts for different execution workflows are also included. run_thermo_mpi.sh runs a 10 picosecond simulation, saves the energy outputs and prints the total, kinetic and potential energies as a functions of time. run_thermo_serial.sh does the same thing with a system of equal size with the serial executable. Also, one is provided for running the executabel of the hybrid version, run_mpi_omp.sh.
There's also a shell script, parallel_timescaling.sh, for running samples of ten simulations with a number of processes increasing from 1 to 6, calculating the average wall clock runtime for each number of processes and plotting the time as a function of number of processes with python. A similar shell script, run_puhti_mpi.sh is also included such that the mpi executable can be run on CSC's hpc clusters by submitting the bash script as a batch job to SLURM. This shell script also uses up to 20 processes to measure the scaling.

## 4 Benchmarking

### 4.1 Environment

Benchmarking was done with both consumer grade and hpc hardware. For consumer grade hardware, a laptop with a chip containing six AMD Ryzen 4000-series cpus was used. I tested this by running the MPI program on 16000 atoms with the number of processes going from 1 to 6. For hpc hardware benchmarking, CSC's Puhti cluster was used. In this environment, up to 20

cpus were used on two system sizes. One of 202500 atoms and another of 10000 atoms. The hybrid version combining MPI and multithreading was only run on consumer hardware due to it having been developed last during a time when the SLURM of CSC's puhti cluster was down. Due to the hardware limit of only six processors, each of which allow only one thread, this version could only be run with limited options for the number of processes and threads. I used 3 MPI processes and assigned two threads to each.

## 4.2 Measurements

It became immediately obvious that the serial version runs significantly slower than the mpi version with multiple processes and that increasing the number of processes generally decreases runtimes. The hybrid version, running 3 concurrent processes, each of which having two threads, also performed substantially more poorly than the MPI version that used 6 processes. In fact, the hybrid version barely even performed better than the plain MPI version that used just 3 processes.
With each number of plain MPI processes, ten repetitions were made and the average wall clock times calculated from those repetitions and used in the wall clock time profile plots. Figure (2) displays this exponential decrease in the runtime w.r.t the number of MPI processes on my consumer grade hardware. For the same analysis with CSC's data center hardware, which used up to 20 processes with no multithreading, see figure (3).

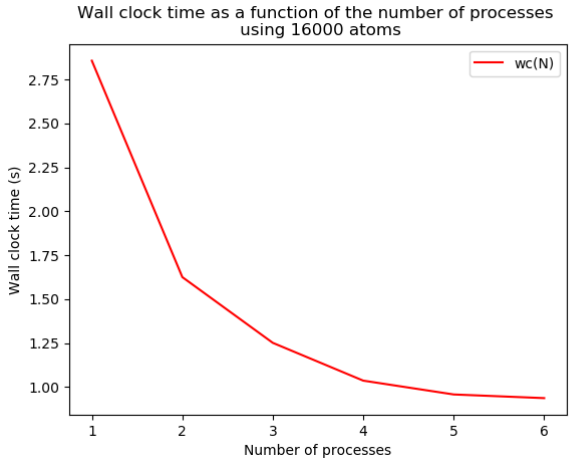


Figure 2: Average wall clock time decreases on consumer grade hardware in an exponential fashion as the number of processes is increased. Using from 1 to up to 6 mpi processes with no multithreading.

By the looks of figures (2), (3) and (4) execution time as a function of

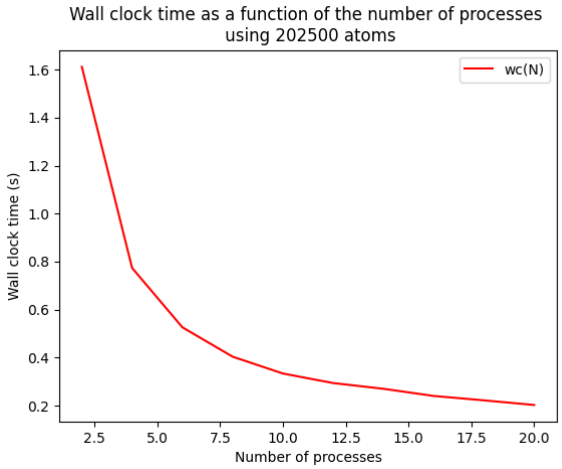


Figure 3: Average wall clock time decreases on CSC's hpc data center hardware Puhti in an exponential fashion as the number of processes is increased. Using from 2 to up to 20 processes with no multithreading.

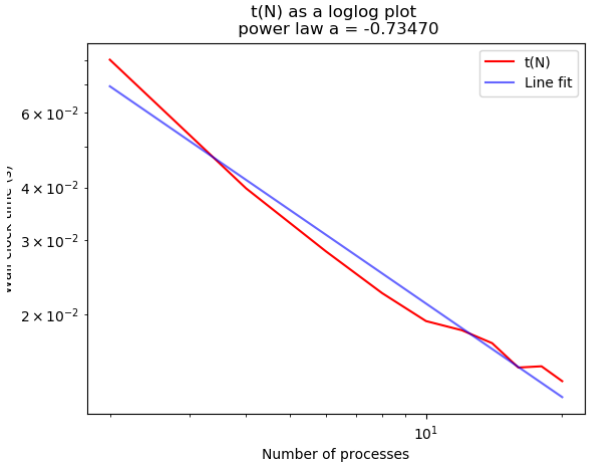


Figure 4: A line fit to a similar loglog graph of the execution time profile w.r.t. the number of processes. The graph seems to roughly abide by power law.

average wall clock time displays an exponential relationship in each case, which was expected. It seems to follow a power law. A linear function was also fitted to the logarithmized data of the curve with python, from which the slope, representing the exponent of the power law, could be extracted. This was done with the powerlaw.py python script and the loglog plot along with the linear fit is shown in figure (4). By calculating the slope, it was observed that the wall

clock time roughly exhibits this power law behaviour,

$$t(N) \propto N^{-0.735} \tag{1}$$

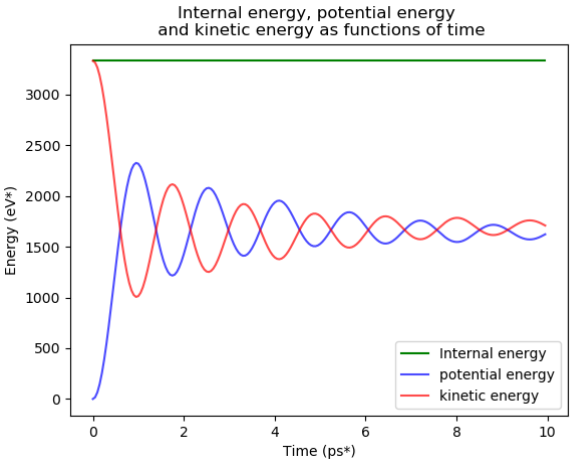


Figure 5: Total, Kinetic and potential energies as functions of time. Total energy stays nicely constant.

# 5 Conclusions

## 5.1 Summary

The parallelization was superbly successful with the MPI standard alone. Surprisingly, the same couldn't exactly be said about the hybrid OpenMP hyperthreading introduced to individual MPI processes. But this smaller endeavor wasn't the main focus of the project, although it could possibly be improved upon with more careful and thorough techniques.
As seen from figure 5, the total energy of the system, which, having subtracted the center of mass velocity, is the internal energy of the system, stays constant, which is expected from an isolated system like this. Likewise, the kinetic and potential energies oscillate back and forth. These suggest that the implementation leads to physically meaningful dynamics.

## 5.2 Possible improvements

Despite very favorable time scaling with plain MPI processes, the hybrid multithreading-with-MPI parallelization techniques could for sure be further investigated and refined, although I personally have marginal confidence in much improvement

over the pure MPI approach emerging from that enterprise. Admittedly, due to its initially poor results in running single runs, I never rigorously did further benchmarking with it on CSC's or the university's data center hardware and instead chose to focus the thorough analysis on the pure MPI program.
Regarding the storage of atom position and neighbor data in the subdomains, a programmatically more intuitive approach could've been to gather all atoms, along with the receive buffers, into an actual 2D array where each row corresponds to a row of physical atoms, instead of having all the rows of the atoms of a subdomain be stored back-to-back to a single 1D array. This could've resulted in more intuitive message-passing programming indexing when exchanging edge atom position data during time integration.
On another note, extending the utilization of hardware from just CPUs to also include allocation of the atom data to GPUs and executing some of the computations there massively in parallel sounds promising and worth trying in the future. For this, CUDA would be a natural choice of platform for NVIDIA hardware like the GPUs on the university's Turso cluster.