University of Antwerp
Faculty of Science

Parallel Programming

Project assignment

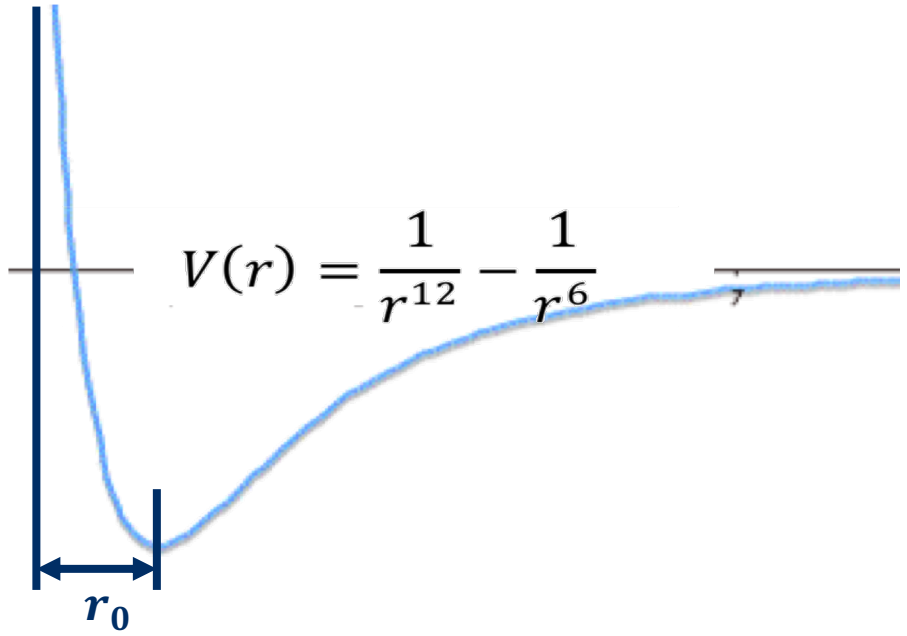
Domain decomposition for Molecular Dynamics

phase 1

sequential code

Molecular Dynamics – LJ potential

- interaction = Lennard-Jones potential (without coefficients)
- cut-off distance: $r_c = 3r_0$



- avoid sqrt when computing the distance: $V(r^2) = \frac{1}{(r^2)^6} - \frac{1}{(r^2)^3}$

Molecular Dynamics – Verlet list

- Verlet list of atom i, L_i is list of all atoms j for which

$$r_{ij} \leq r_{cutoff} \text{ or } r_{ij}^2 \leq r_{cutoff}^2$$

- and

$$i < j$$

- this excludes self-interaction ($i = j$), and avoids counting interactions twice ($r_{ij} = r_{ji}$)
- We stick to the two-dimensional case
  - easier to visualise

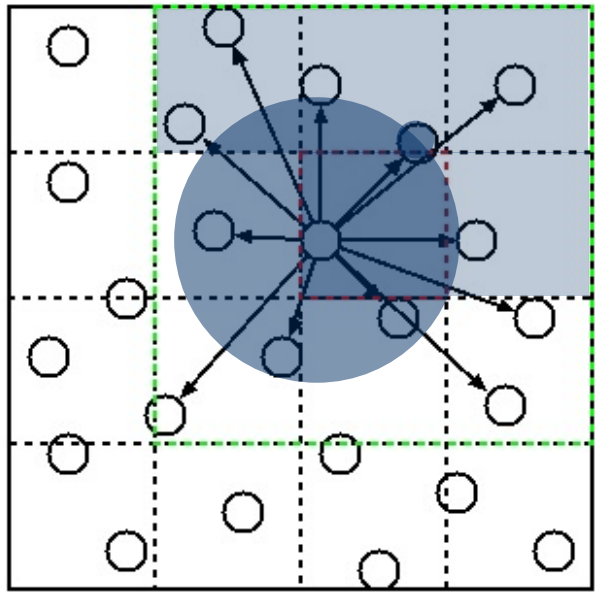
Molecular Dynamics

- Create a square grid, with grid size r_c
- Cell C_{kl} is list of atoms i for which

$$kr_c \leq x_i < (k+1)r_c$$

$$lr_c \leq y_i < (l+1)r_c$$

- Denote the cell of atom i as $C(i)$
- To build the Verlet list of atom i in $C(i) = C_{kl}$ you loop over all atoms $j \neq i$ in $C_{k,l}$, $C_{k,l+1}$, $C_{k-1,l+1}$, $C_{k,l+1}$, and $C_{k+1,l+1}$.
- This way, every pair is visited only once!



Molecular dynamics – force and acceleration

- **The force exerted by atom j on atom i is**

$$\bar{F}_{ij} = \nabla V(r_{ij}) = \frac{dV(r^2)}{dr^2}\nabla(r^2) = \frac{dV(r^2)}{dr^2} \cdot 2r \cdot \bar{r}$$

- **and**

$$\bar{F}_{ji} = -\bar{F}_{ij}$$

- **total force on atom i:**

$$\bar{F}_i = \sum_{j \in L_i} \bar{F}_{ij}$$

- **acceleration:**

$$\bar{a}_i = \frac{\bar{F}_i}{m_i}$$

Molecular dynamics – time evolution

- **time integration scheme**
- **time step = δt**

- Velocity Verlet Algorithm:
  - Each integration cycle
  1. Calculate velocities at mid-step $v(t+\frac{\delta t}{2}) = v(t) + \frac{1}{2}a(t)\delta t$
  2. Calculate positions at the next step $r(t+\delta t) = r(t) + v\left(t+\frac{\delta t}{2}\right)\delta t$
  3. Calculate accelerations at next step from the potential
  4. Update the velocities $v(t+\delta t) = v\left(t+\frac{\delta t}{2}\right) + \frac{1}{2}a(t+\delta t)\delta t$

Molecular dynamics – time evolution

- **Atoms may move from one cell to another. One must update the grid cells every time the Verlet list must be updated.**

Molecular Dynamics - initialization

- you must control the location of atoms at $t = 0$. If, accidentally, two atoms are very close, the force will be very high and the atoms will move too fast, your simulation will explode. If they are too far the the force will be small and the atoms move slowly
- fixed timestep is problematic, select timestep as function of highest force or acceleration
- generate atoms in close packing (hexagonal in 2D, FCC in 3D), add a bit of noise to the positions
- This component will be provided by me

Strategy

- **make sure you have carried out personal-setup.pdf**
  - github account!
  - mpi installation (was forgotten in the VM)
- **see principles in first talk**
- **use micc**
  - put your code in a private github repository and give me access
    - you have a backup at any time (commit and push regularly)
    - I can check your code and investigate your problems to help you
    - the code is a deliverable for the evaluation

Strategy

- **start in Python**
- **write tests for every component**
  - LJ potential and force
  - grid construction
  - verlet list construction
  - total forces
  - time evolution
- **replace Python version with a C++ or Fortran version for**
  - LJ potential and forces
  - time evolution
  - (other components may remain in Python even if they are slow)
  - provide timings for both versions (e.g. using the et_stopwatch package, see micc tutorials)

phase 2

Parallellisation

Molecular Dynamics - Parallelization

- **Divide the work across several processes, using mpi4py. Every core runs a separate process that does part of the work. E.g.**

  ```
  > mpirun –np 2 python md.py
  ```

  runs 2 processes. If the tasks are independent, they must not exchange information. This is not always possible.
- **Check the (api) documentation of mpi4py at https://mpi4py.readthedocs.io/en/stable/**
- **For a tutorial on MPI check https://folk.idi.ntnu.no/elster/tdt4200-f09/gropp-mpi-tutorial.pdf**

Molecular Dynamics - Parallelization

- **domain composition**
  - Divide the space in domains, one per process. Each domain has its own atoms.
  - Let domain boundaries coincide with cell boundaries.
  - Domain needs information from neighbouring domains. Cells on the boundary must be known to each domain touching that boundary in order to compute the Verlet list. This is the halo concept, also known as ghost cells.
  - Atoms may move from one cell to another and from one domain to another.
  - Think carefully about which information must be communicated and when
  - The amount of communication is proportional to the area of the boundaries
  - The amount of work (load) is proportional to the volume of the domains

Molecular Dynamics - Parallelization

- **Communication is slow and is not useful work on its own**
- **In general we want to minimize the amount of data communicated.**
- **Try to overlap computation and communication. While waiting for the communication to finish, do some other useful computation.**
  - check the tutorial by Gropp and make sure you understand these concepts:
    - one-sided vs two-sided communication
    - blocking vs non-blocking communication

Molecular Dynamics - Parallelization

- **if the atom density is uniform across domains, each process has approximately the same amount of work and the processes will finish their timestep more or less simultaneously.**
- **If not, some processed will finish their timestep well before others and will have to wait for the last process before the next timestep can be started. Load balancing needed**
  - Dynamical domain decomposition. E.g. using only boundaries perpendicular to the x-axis, letting the boundaries move along the x-axis so they have approximately the same atom density.
  - Much more domains (= small tasks) than processes. The tasks are moved around to waiting processes.
  - We won't go there…

Molecular Dynamics - Parallelization

- **start simple**
  - small domain, no cut-off, no time integration, compute the interaction energy.
  - add time integration
  - add cut-off (brute force $O(N^2)$)
- **Divide the space in two domains along a straight boundary parallel with cell boundaries.**
- **think about the consequences of having the domain boundary in the middle of the cells vs coinciding with the cell boundary.**
- **decide what to communicate and how**
- **try to overlap computation and communication**
- **Once you understand the machinery, you can proceed further**
  - you should get a 2 domain case working
  - draw up a plan on how to go beyond 2 domains

phase 3

Evaluation

Evaluation

- **presentation of your work (probably via ms teams)**
  - explain your approach
  - evaluate the performance of your approach (measurements)
    - (I will give you some tools later)
  - I will ask questions to see if you understand what you did, and to challenge your approach
- **deliverables**
  - the presentation (.pptx or .pdf)
  - the code
  - both should be in your github repository

Getting started

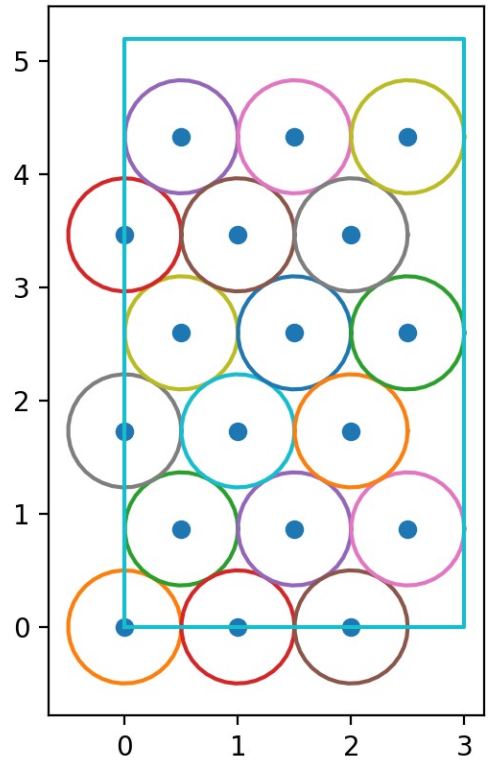
Getting started

```
# create project
> cd workspace
> micc create LeonMD --package --remote='private'
> cd LeonMD
# add common tools for atom generation and plotting
> poetry add et_ppMDcommon
> source .venv/bin/activate
(.venv)>
```

Getting started

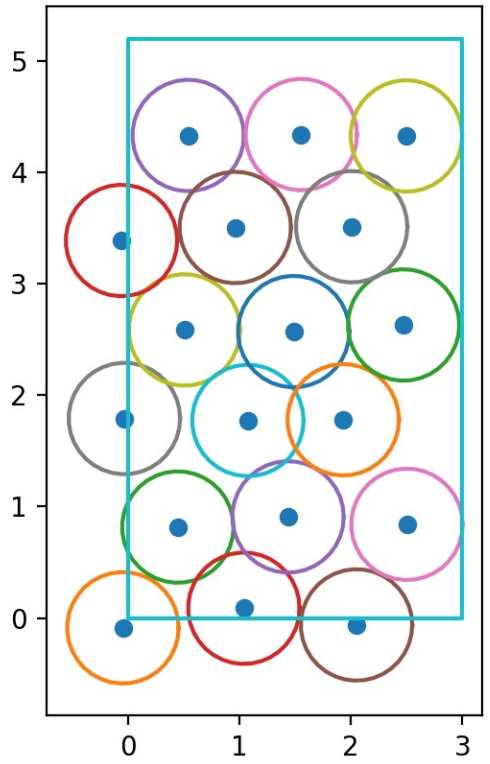
```
import numpy as np
import et_ppmdcommon as cmn

# make box containing 3x3 rectangular unit cells
box = cmn.Box(0, 0, 3, 3*np.sqrt(3))
x,y = box.generateAtoms(r=1.0)
md.figure()
md.plotAtoms(x, y)            # dots
md.plotAtoms(x, y, r=r0/2)    # circles with atom radius
md.plotBox(box)               # containing box
md.plt.show()
```



Getting started

```
x,y = box.generateAtoms(box, r=1.0, noise=.1)
md.figure()
md.plotAtoms(x, y)              # dots
md.plotAtoms(x, y, r=r0/2)      # circles with atom radius
md.plotBox(box)                 # containing box
md.plt.show()
```



equations

LJ force

$$\frac{d}{dr}(r^{-12} - r^{-6})\frac{dr}{dx} = (-12\,r^{-13} + 6r^{-7})\frac{d}{dx}(x^2 + y^2)^{1/2}$$

$$= (-12\,r^{-13} + 6r^{-7})\frac{1}{2}(x^2 + y^2)^{-1/2}\frac{d}{dx}(x^2 + y^2)$$

$$= (-12\,r^{-13} + 6r^{-7})\frac{1}{2r}2x = (-12\,r^{-14} + 6r^{-8})x$$

$$= (6r^{-8} - 12\,r^{-14})x$$

$$\bar{\boldsymbol{F}}_{\boldsymbol{ij}} = \boldsymbol{\nabla V}(\boldsymbol{r}_{\boldsymbol{ij}}) = (6r^{-8} - 12\,r^{-14})\bar{r}$$

$$= 6r^{-8}(1 - 2\,r^{-6})\bar{r}$$

$$= 6(r^2)^{-4}(1 - 2(r^2)^{-3})\bar{r}$$

LJ equilibrium distance

$$\frac{d}{dr}(r^{-12} - r^{-6}) = (-12\, r^{-13} + 6r^{-7})$$

$$= 6(1 - 2\, r^{-6})r^{-7}$$

$$= 6\left(1 - \frac{2}{r^6}\right)r^{-7} = 0$$

$$\Leftrightarrow r = 2^{\frac{1}{6}} \cong 1.122462048309373$$

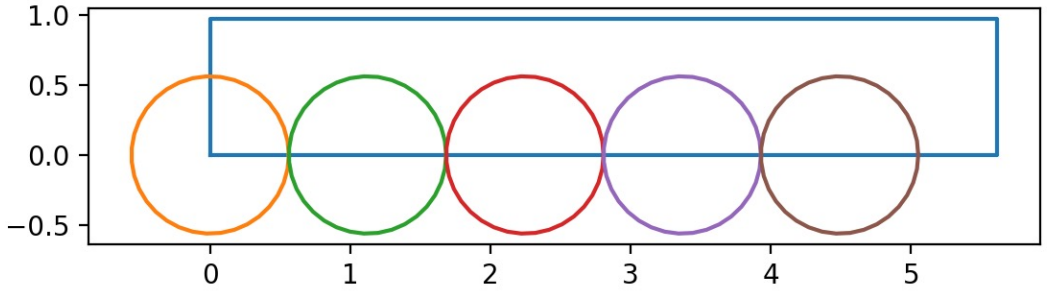
Verlet Lists

Verlist list

- **is a mathematical concept: the set of neighbours of an atom which are within the cutoff distance**
- **in a program you need a representation for each mathematical concept:**
  - a data structure
  - the data structure is **not** important for correctness
  - the data structure is important for
    - performance
    - practical considerations (ease of programming, debugging, …)

Pure Python approach

- $r_{cutoff} = 2.5R_0$
- **a list of lists**
- **vl = [[1,2],[2,3],[3,4],[4]]**
- **advantages:**
  - simple, good as first approach
  - lists are dynamic
  - verlet list of atom i is vl[i]
  - n_atoms = len(vl)
  - n_neighbours_i = len(vl[i])



- `vl = []`
  `[]`
- `vl0 = []`
  `[]`
- `vl0.append(1)`
  `[1]`
- `vl0.append(2)`
  `[1,2]`
  `vl.append(vl0)`
  `[[1,2]]`

- `vl1 = []`
  `[]`
- `vl1.append(2)`
  `[2]`
- `vl0.append(3)`
  `[2,3]`
- `vl.append(vl1)`
  `[[1,2],[2,3]]`
- …

mixed programming approach

- **disadvantages**
  - double loop in Python = slow
  - Python list is non-contiguous data structure = slow
- **replace with contiguous data-structure: numpy array**
  - pass this to C++ or Fortran and make the loops in C++/Fortran

Atom	#neigbours	Neighbour 1	Neighbour 2	Neighbour 3
0	2	1	2	-
1	2	2	3	-
2	2	3	4	-
3	1	4	-	-
4	0	-	-	-

Atom	#neigbours	Neighbour 1	Neighbour 2	Neighbour 3
0	2	1	2	-
1	2	2	3	-
2	2	3	4	-
3	1	4	-	-
4	0	-	-	-

- **if storage is row major (rows are stored contiguously)**
  - Python, C, C++
  - outer loop = loop over atoms (loop over rows
  - inner loop = loop over neighbours (loop in a row) => contiguous data access
- **if storage is column major (columns are stored contiguously)**
  - transpose the data structure
  - outer loop = loop over atoms (loop over columns)
  - inner loop = loop over neighbours (loop in a column) => contiguous data acces

Atom	#neigbours	Neighbour 1	Neighbour 2	Neighbour 3
0	2	1	2	-
1	2	2	3	-
2	2	3	4	-
3	1	4	-	-
4	0	-	-	-

- **disadvantages**
  - memory space wasted
  - cache misses
  - need to transpose to switch between Fortran and C++
  - Fortran requires multi-dimensional numpy arrays to be created with ‘Fortran’ storage order
    - vl = numpy.zeros((max_neighours, n_atoms), dtype=float, order=‘Fortran’)
    - if not, the array will be copied and transposed, which is expensive
  - consequently, our Python coded will be different depending on whether we want to use Fortran or C++
    - that may complicate things later

Atom	#neigbours	Neighbour 1	Neighbour 2	Neighbour 3
0	2	1	2	-
1	2	2	3	-
2	2	3	4	-
3	1	4	-	-
4	0	-	-	-

- **Linearize the numpy array**

2	1	2	2	2	3	2	3	4	1	4	0

  - length = n_atoms (5) + total number of neighbours (7). = 12
- **Verlet list is reused often before it is rebuilt, so it is a good thing to optimize.**
- **1D data arrays can be passed to Fortran and C++ regardless of the storage order, because they are contiguous anyway.**