A Domain Specific Language for Performance Portable Molecular Simulations

William Robert Saunders, James Grant, Eike Hermann Müller* University of Bath, United Kingdom * poster presenter

Challenges for modern MD codes

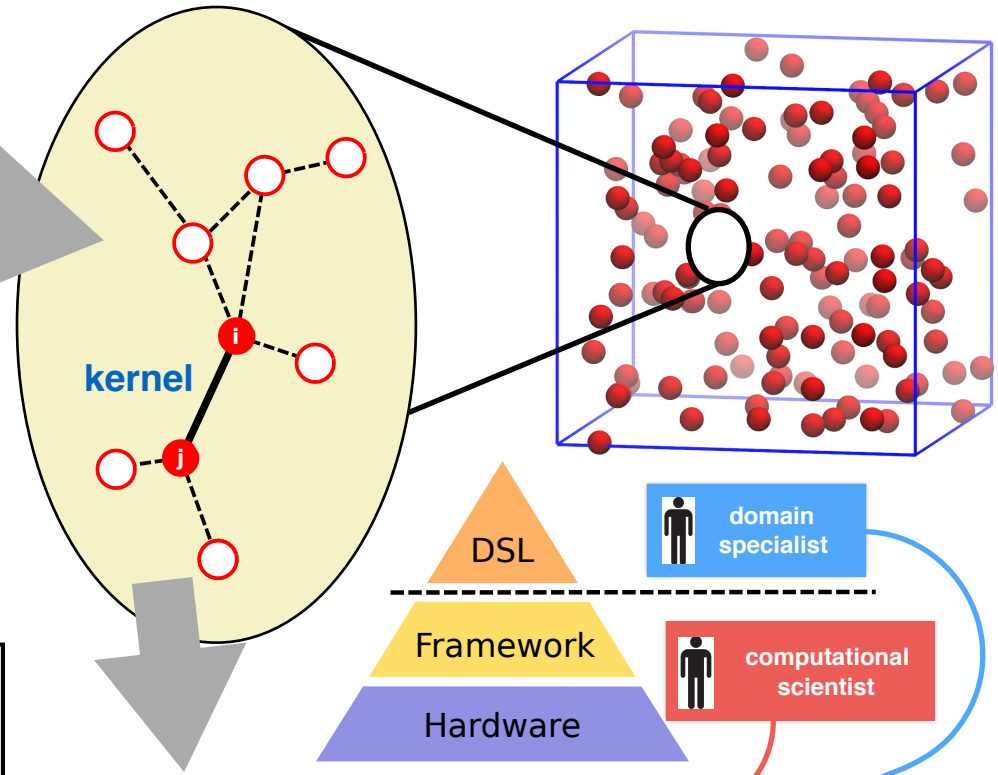
Molecular Dynamics (MD) codes predict the physical properties of matter by simulating a large number of interacting particles. To make accurate predictions, models need to run efficiently on massively parallel computers and manycore chips.

This creates several challenges:

- diversifying hardware landscape
- parallelisation/optimisation often inseparable from science code
- growing data volumes, (post-) processing (=structure analysis) often sequential
- development of efficient code requires diverse range of skills:
  1. domain specific knowledge (physics/chemistry)
  2. parallel programming & optimisation on CPUs, GPUs, Xeon Phi,…
     - ➡ significant manpower required
     - ➡ rare for an individual to possess all those skills
     - ➡ rewrite code for each new hardware platform

Our design principle: **"separation of concerns"** between *domain specialist* and *computational scientist*

➠ suitable **abstraction?**



Python code generation system

- Python code generation framework for Molecular Dynamics Simulations
- algorithms (timestepping, thermostats,…) expressed at high abstraction level
- computationally expensive loops over particles and particle pairs are realised as auto-generated C-code
- domain specialist only has to write Python algorithm and local particle- (pair-) kernels
- particle properties with access descriptors determine required parallel operations outside kernel call (compare PyOP2 [1] for grid-based PDE solvers)
- flexibility and convenience of high-level language combined with performance of compiled code

source code
`https://bitbucket.org/wrs20/ppmd`

Key idea & abstraction

fundamental operation in MD codes:

```
for all particle pairs (i,j):
   execute user-defined kernel
```

- pair looping hardware dependent (neighbour-list, layer algorithm, …)
- includes non-force calculation (local analysis kernels)
- fundamental data structures:
  1. particle properties (mass, position, velocity,…)
  2. global properties (total energy, RDF, …)

➠ **Separate algorithm and kernel from parallel loop over pairs**

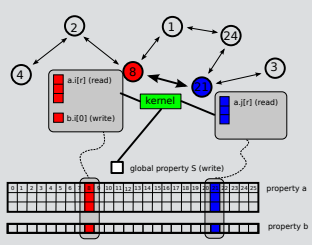
Data structures & execution model

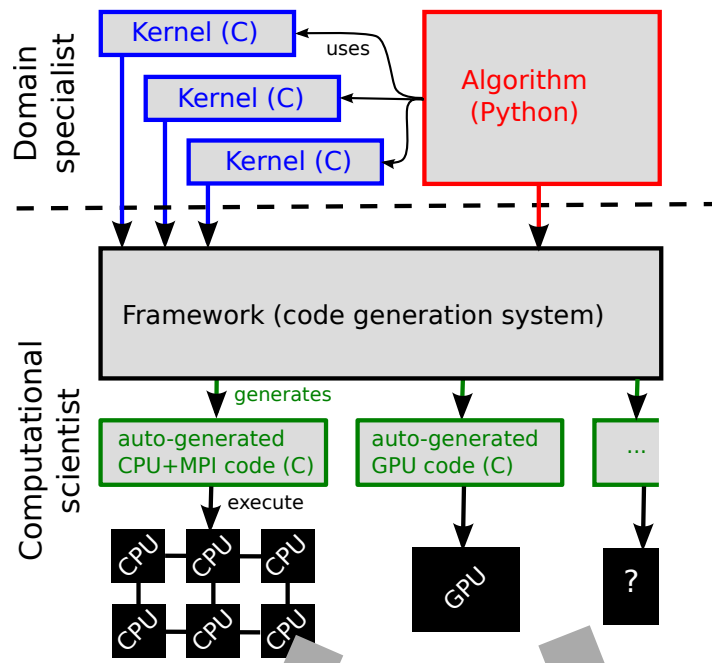
Particle properties (mass, position, velocity)

- stored in a numpy array wrapped in a Python `ParticleDat` object
- r-th component of property a for particle pair (i,j) accessible as `a.i[r]` and `a.j[r]` in C-kernel
- storage on host or device memory, access marks data as dirty (for halo exchanges) and/or triggers copies between host and device
- global properties (total energy, RDF) stored as `ScalarArray` objects

Particle pair loops

- executes C-kernel over all particle pairs (i,j)
- access descriptors trigger halo exchanges, if necessary
- C-code for pair-looping on a particular architecture is auto-generated and kernel inlined
- currently support for MPI, CUDA, MPI+CUDA
- particle list, neighbour-list, neighbour-matrix





Example code

Given particle property a, calculate particle property b and global property S^g according to:

$$b^{(i)} = \sum_{\text{all pairs } (i,j)} \sum_{r=0}^{d-1} \left(a_r^{(i)} - a_r^{(j)}\right)^2$$

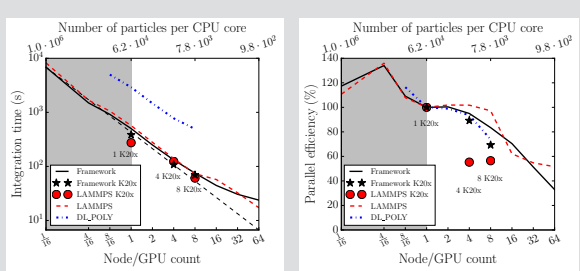
$$S^g = \sum_{\text{all pairs } (i,j)} \sum_{r=0}^{d-1} \left(a_r^{(i)} - a_r^{(j)}\right)^4.$$

```
dim=3 # dimension
npart=1000 # number of particles

# Define Particle Dats
a = ParticleDat(npart=npart, ncomp=dim)
b = ParticleDat(ncomp=1,npart=npart, initial_value=0.0)
S = ScalarArray(ncomp=1,initial_value=0.0)

kernel_code='''
  double da_sq = 0.0;
  for (int r=0;r<dim;++r) {
    double da = a.i[r]-a.j[r]; da_sq += da*da;
  }
  b.i[0] += da_sq; S += da_sq*da_sq;
'''

# Define constants passed to kernel
consts = (Constant('dim', dim),)

# Define kernel
kernel = Kernel('update',kernel_code,consts)

# Define and execute pair loop
pair_loop = PairLoop(kernel=kernel, {'a':a(access.READ),
                                     'b':b(access.INC),
                                     'S':S(access.INC)})
pair_loop.execute()
```

Results I strong scaling/comparison to monolithic MD codes

- compare to LAMMPS, DL_POLY
- Lennard-Jones benchmark
- 1 million atoms
- Kepler K20X GPU & Intel Xeon E5-2650v2 CPU (Ivybridge)
- 64 CPU nodes (1024 cores)

total performance	% of peak	% of time
16 core Intel Xeon (333 GFLOPs)	16.5%	54.8%
nVidia K20X GPU (1310 GFLOPs)	11.9%	36.9%



Results II structure analysis

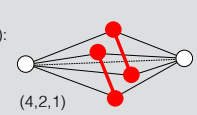
- LJ-run with bond angle analysis (BOA) [3]

$$Q_\ell^{(i)} = \sqrt{\frac{4\pi}{2\ell+1}\sum_{m=-\ell}^{+\ell}\left|q_{\ell m}^{(i)}\right|^2}, \quad q_{\ell m}^{(i)} = \frac{1}{N_{\text{nb}}}\sum_{j=0}^{N_{\text{nb}}-1} Y_\ell^m\left(\boldsymbol{r}^{(i)} - \boldsymbol{r}^{(j)}\right)$$

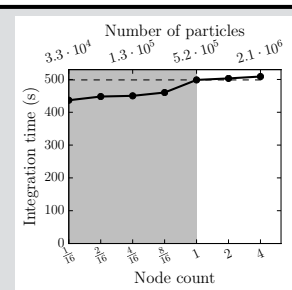
- Common neighbour analysis (CNA) [4] in post processing stage

classify pairs by triplet (n_{nb},n_b,n_{lcb}):

1. # common neighbours
2. # neighbour links
3. neighbour cluster size



(4,2,1)



weak scaling
Lennard-Jones with on-the-fly BOA

[1] Rathgeber et al.: *"PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes."* In HPC, Networking Storage and Analysis, SC Companion, Los Alamitos, CA, USA, 2012. IEEE Computer Society
[2] Saunders, Grant and Müller: *"A Domain Specific Language for Performance Portable Molecular Dynamics Algorithms"* submitted to Computer Physics Communications [arxiv:1704.03329]
[3] Steinhardt, Nelson, and Ronchetti: *"Bond-orientational order in liquids and glasses"*. Physical Review B, 28(2), p.784. 1983
[4] Honeycutt and Andersen: *"Molecular dynamics study of melting and freezing of small Lennard-Jones clusters"*. J. Phys. Chem., 91(19), pp.4950-4963. 1987
This research made use of the Balena High Performance Computing Service at the University of Bath. The PhD project of William Saunders is funded by an EPSRC studentship