A Performance Portable Framework for Molecular Simulations

William Saunders*, James Grant†, **Eike Hermann Mueller***

University of Bath
*Mathematical Sciences, †Chemistry

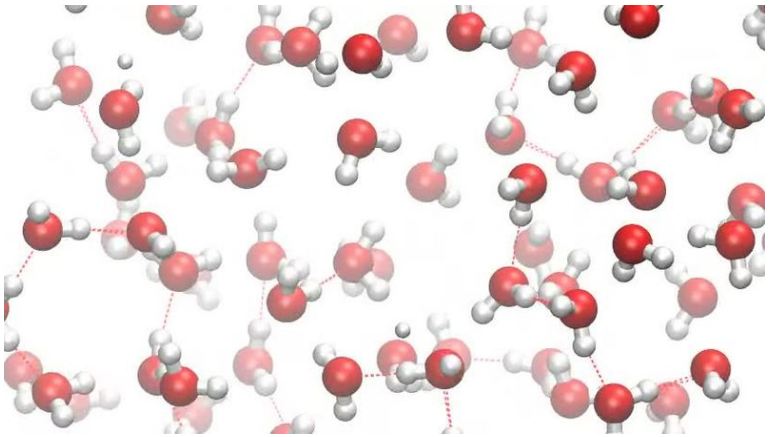
[Comp. Phys. Comms. (2018) vol 224, pp. 119-135] and [arXiv:1708.01135]

Scientific Computing Seminar,
University of Warwick, Mon 18th June 2018

Overview

Molecular Simulation

Molecular Simulation codes are major HPC users



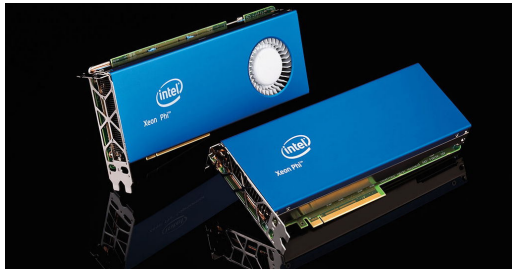
ARCHER top 10 codes

1	VASP	16.8%
2	**Gromacs**	8.3%
3	CASTEP	5.0%
4	**cp2k**	4.8%
5	Q. Espresso	4.2%
6	HYDRA	3.8%
7	**LAMMPS**	3.7%
8	OpenFOAM	3.2%
9	**NAMD**	3.1%
10	WRF	2.1%

$\approx$ 20% of time spent on molecular particle integration

(and even more on quantum chemistry)

The hardware zoo



Source: Wikipedia, Flickr, CC license

- Several layers of parallelism
  - Distributed memory across nodes (MPI)
  - Shared memory on a node (OpenMP, threads, MPI, Intel TBB)
  - Vectorisation (CUDA, intrinsics, vector libraries)
- Complex memory hierarchy

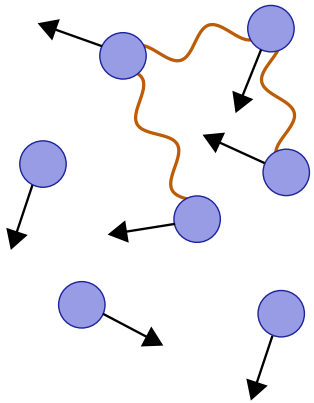
$\Rightarrow$ **Porting MD codes is hard**

Structure analysis codes often handwritten and not parallel

Molecular Dynamics

Simulating Particles

Follow trajectories of a large number ($\gtrsim 10^6$) interacting particles



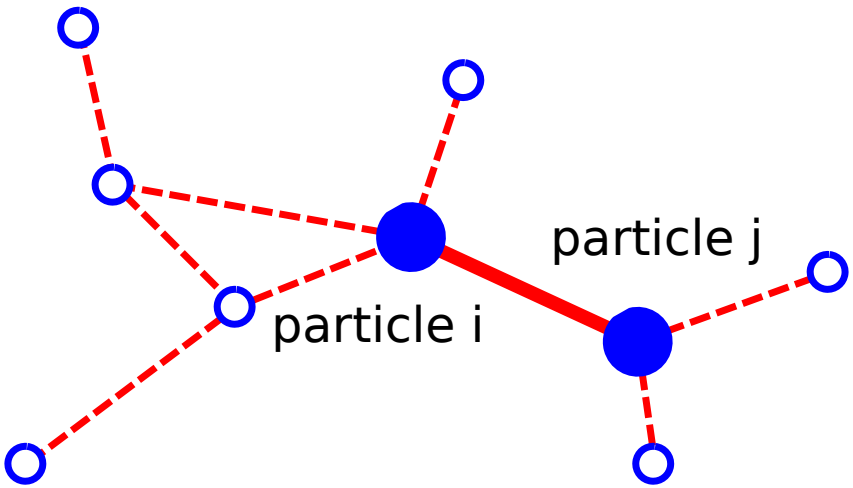
$$m_i \frac{d\boldsymbol{v}^{(i)}}{dt} = \boldsymbol{F}^{(i)}(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)}),$$
$$\frac{d\boldsymbol{x}^{(i)}}{dt} = \boldsymbol{v}^{(i)} \qquad \text{for } i = 1, \ldots, N$$

Key operations

- Local updates $\boldsymbol{x}^{(i)} \mapsto \boldsymbol{x}^{(i)} + \delta\boldsymbol{x}^{(i)} = \boldsymbol{x}^{(i)} + \delta t \cdot \boldsymbol{v}^{(i)}$
- Force calculation $\boldsymbol{F}^{(i)} = \boldsymbol{F}^{(i)}(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)})$
- Query local environment (structure analysis)

Abstraction

"For all pairs of particles do operation $X_{\text{pair}}(i,j)$"



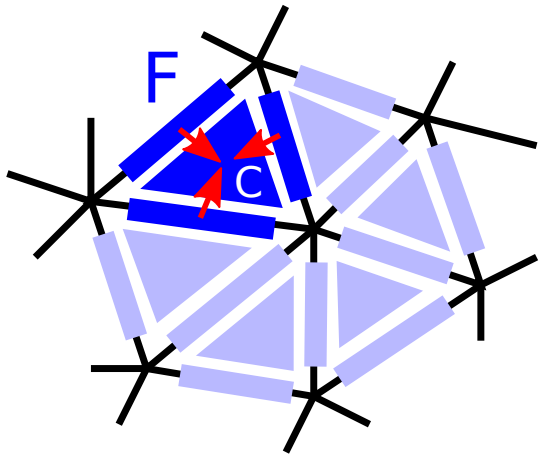
"For all pairs with $|\boldsymbol{r}^{(i)} - \boldsymbol{r}^{(j)}| < r_c$ do operation $X_{\text{local pair}}(i,j)$"

How this is executed is

- Hardware specific
- Of no interest to the scientist (domain specialist)

Grid iteration in PDE solvers

Inspired by the **(Py)OP2** library [Rathgeber et al. (2012)]



for all cells C **do**
  for all facets F of C **do**
    $a_C \mapsto a_C + \rho \cdot b_F$
  end for
end for

Python/C Source code

```
# Define Dats
a = op2.Dat(cells)
b = op2.Dat(facets)
# Local kernel code
kernel_code='''
void flux_update(double *a,
                 double **b) {
  for (int r=0;r<4;++r)
    a[0] += rho*b[r][0];
}'''
# Define constant passed to kernel
rho = op2.Const(1, 0.3, name="rho")
# Define kernel
kernel = op2.Kernel(kernel_code)
# Define and execute pair loop
par_loop = op2.ParLoop(kernel,cells,
  {'a':a(op2.INC),
   'b':b(op2.READ,facet_map)})
```

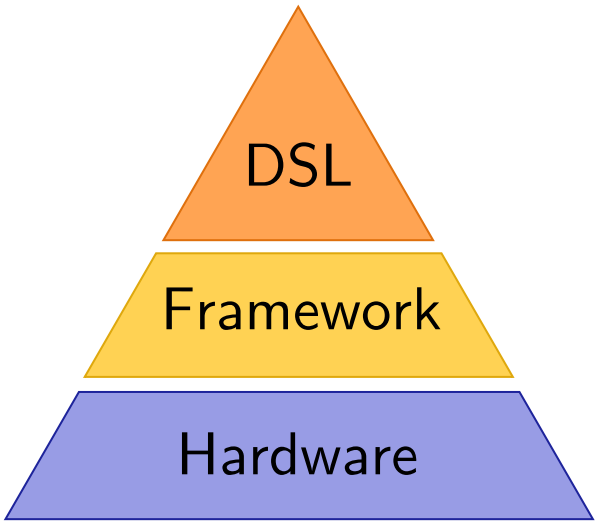
Grid iteration/paralellisation hidden from user!

Abstraction

Key idea: "Separation of Concerns"

- Domain specialist:
  - Local particle-pair kernel
  - Overall algorithm

  $\Rightarrow$ **hardware independent DSL**
- Computational scientist:
  Framework to execute kernel over all pairs on a particular hardware

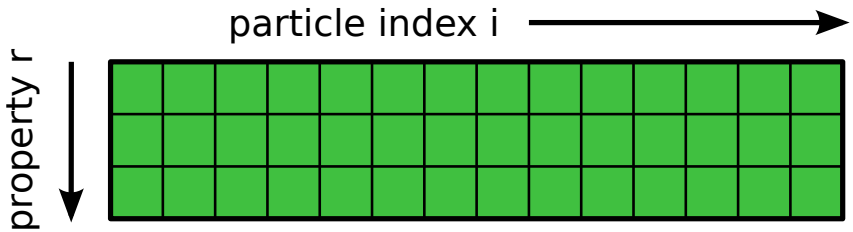


Data structures

Each particle $i = 1, \ldots, N$ can have $r = 1, \ldots, M$ properties $a_r^{(i)}$
e.g.

- Mass
- Position, Velocity
- # of neighbours within distance r_c

Store as 2d numpy arrays wrapped in Python `ParticleDat` objects
Access as `a.i[r]` and `a.j[r]` in kernel $X_{\text{pair}}(i, j)$



M^g Global properties S_r^g

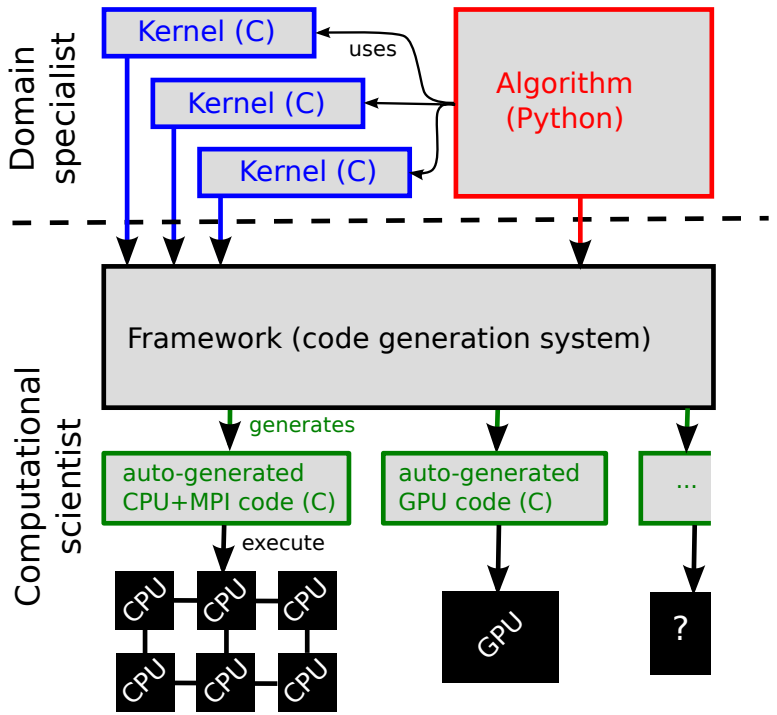
- Energy
- (binned) Radial Distribution Function (RDF)

Store as 1d numpy arrays wrapped in Python `ScalarArray` objects

Execution model

Python Code-generation system

- Domain specialist writes small C-kernel which describes $X_{\text{local pair}}(i, j)$
- Access descriptors (`READ`, `WRITE`, `INC`, ...)
- auto-generated C-wrapper code for kernel execution
- Necessary parallelisation calls inserted, based on access descriptors



Example

Input

- Particle property $\boldsymbol{a}$

  (vector valued)

Output

- Particle property b
- Global property S^g

$$b^{(i)} = \sum_{\text{pairs } (i,j)} \left\| \boldsymbol{a}^{(i)} - \boldsymbol{a}^{(j)} \right\|^2$$

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

$$S^g = \sum_{\text{pairs } (i,j)} \left\| \boldsymbol{a}^{(i)} - \boldsymbol{a}^{(j)} \right\|^4$$

Python/C Source code

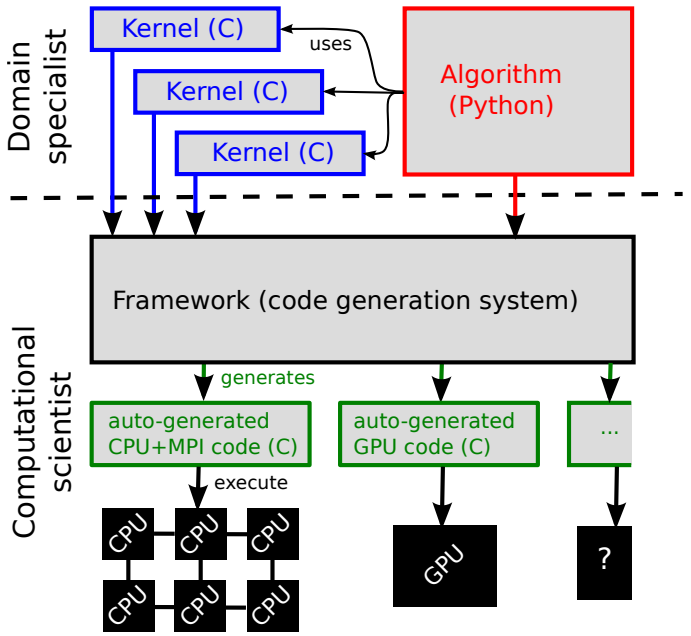
```
dim=3 # dimension
npart=1000 # number of particles
# Define Particle Dats
a = ParticleDat(npart=npart,ncomp=dim)
b = ParticleDat(ncomp=1,npart=npart,
                        initial_value=0)
S = ScalarArray(ncomp=1,initial_value=0)
kernel_code='''
  double da_sq = 0.0;
  for (int r=0;r<dim;++r) {
    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()
```

High-level interface

Framework structure (again)

1. High-level algorithms (timestepper, thermostat, MC sampling, . . .) implemented in Python
2. Local kernel efficiently executed over all pairs

- User doesn't see parallelisation
- ⇒ performance portability

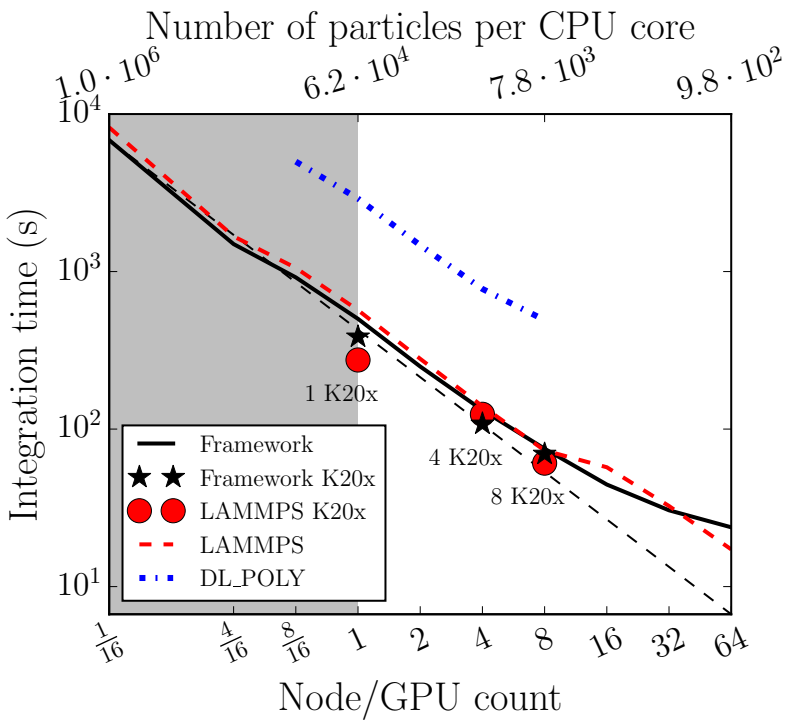


Not just a Python scripting driver layer for existing MD backend

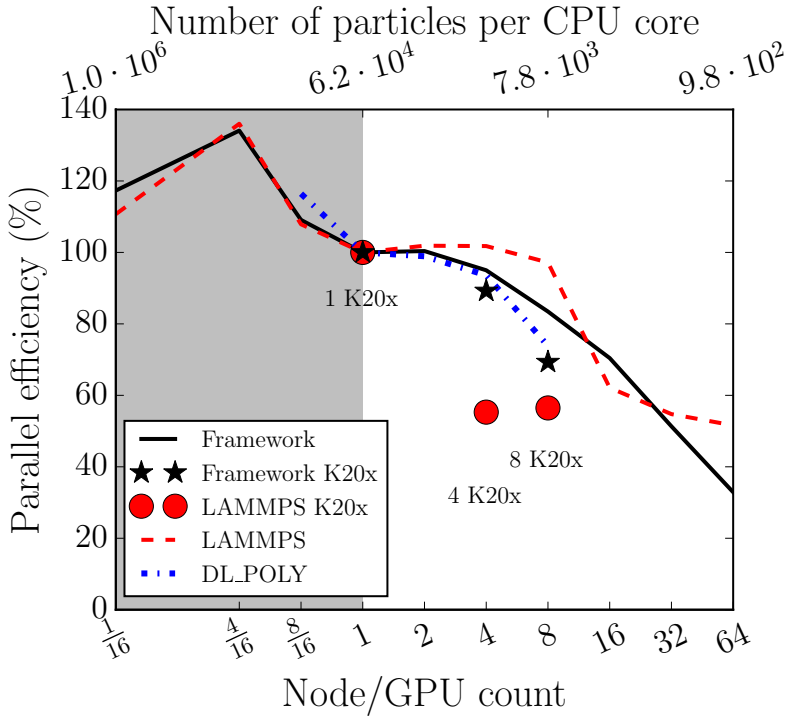
Results I: Scalability

Strong scaling on Balena Lennard-Jones benchmark

10^6 particles, compare to DL_POLY and LAMMPS on CPU and GPU



Solution time



Parallel efficiency

Structure analysis

Not restricted to force calculation

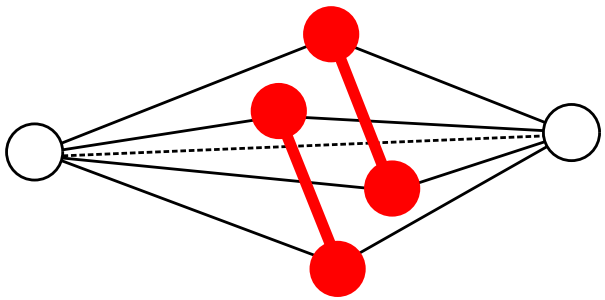
- Bond Order Analysis [Steinhardt et al. (1983)]

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

- Common Neighbour Analysis [Honeycutt & Andersen (1987)]

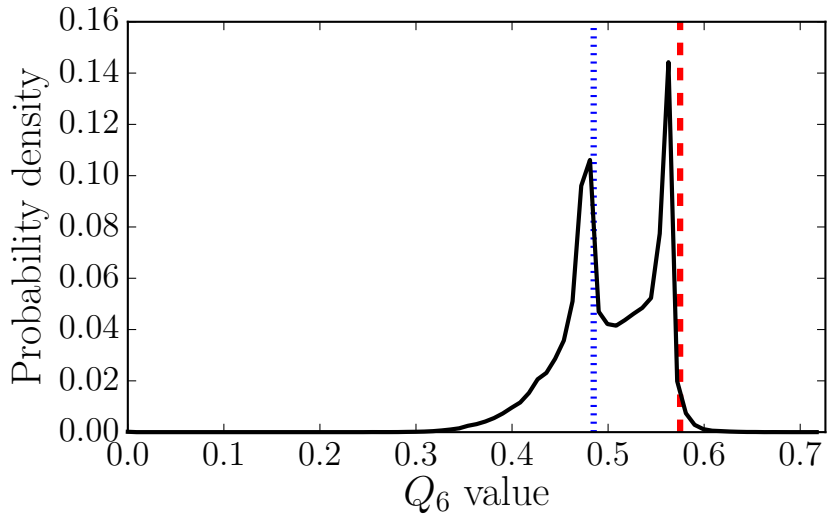
  Classify pairs by triplet $(n_{\mathrm{nb}}, n_{\mathrm{b}}, n_{\mathrm{lcb}})$

  - # common neighbours n_{nb}
  - # neighbour links n_{b}
  - neighbour cluster size n_{lcb}

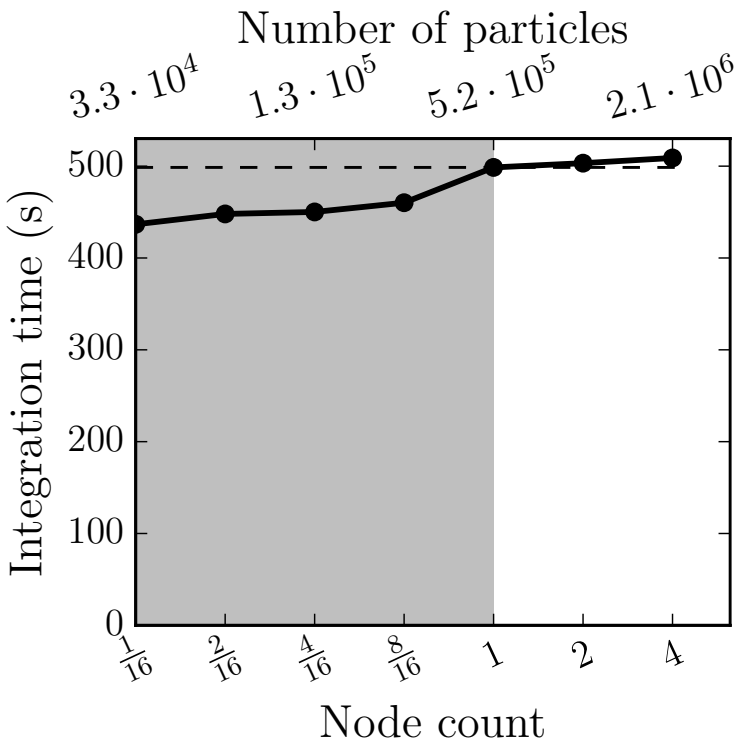


Results II: Structure Analysis

On-the-fly Bond order analysis



Distribution of Q_6



Parallel scalability

Long range Interactions

Hang on, **electrostatics isn't cheap!**

BUT

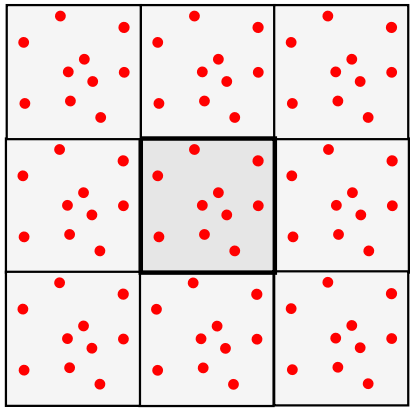
well-defined potential $\propto 1/r$

$\Rightarrow$ only needs to be **implemented once**

Long Range Interactions

Electrostatic potential

$$\phi(\boldsymbol{r}) = \sum_{j=1}^{N} \frac{q_j}{|\boldsymbol{r} - \boldsymbol{r}_j|}$$



Can not truncate

(consider potential of particle in constant charge background)

$\Rightarrow$ Interactions between all particle pairs $\mathcal{O}(N^2)$

What about periodic BCs*?

Three common approaches

1. **Ewald summation**: $\mathcal{O}(N^{3/2})$
2. Smooth Particle Mesh Ewald (SPME): $\mathcal{O}(N \log N)$
3. **Fast Multipole Method (FMM)**: $\mathcal{O}(N)$

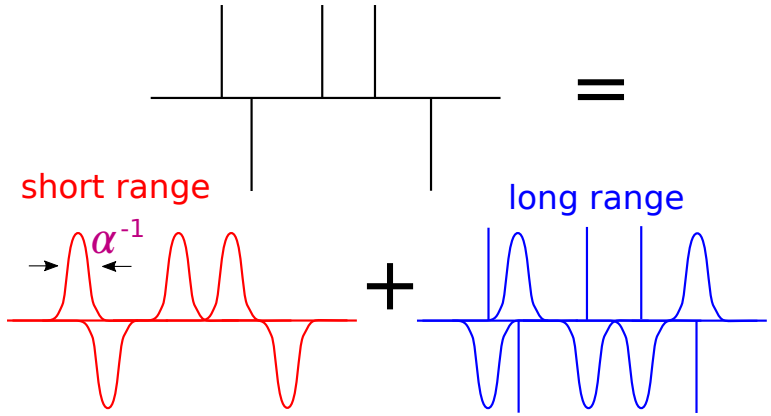
*or mirror charges in Dirichlet/Neumann BCs

Ewald summation

Split charge density and potential [Ewald (1921)]

$$\rho(\boldsymbol{r}) = \rho^{(sr)}(\boldsymbol{r}) + \rho^{(lr)}(\boldsymbol{r})$$

$$\Rightarrow \quad \phi(\boldsymbol{r}) = \phi^{(sr)}(\boldsymbol{r}) + \phi^{(lr)}(\boldsymbol{r})$$



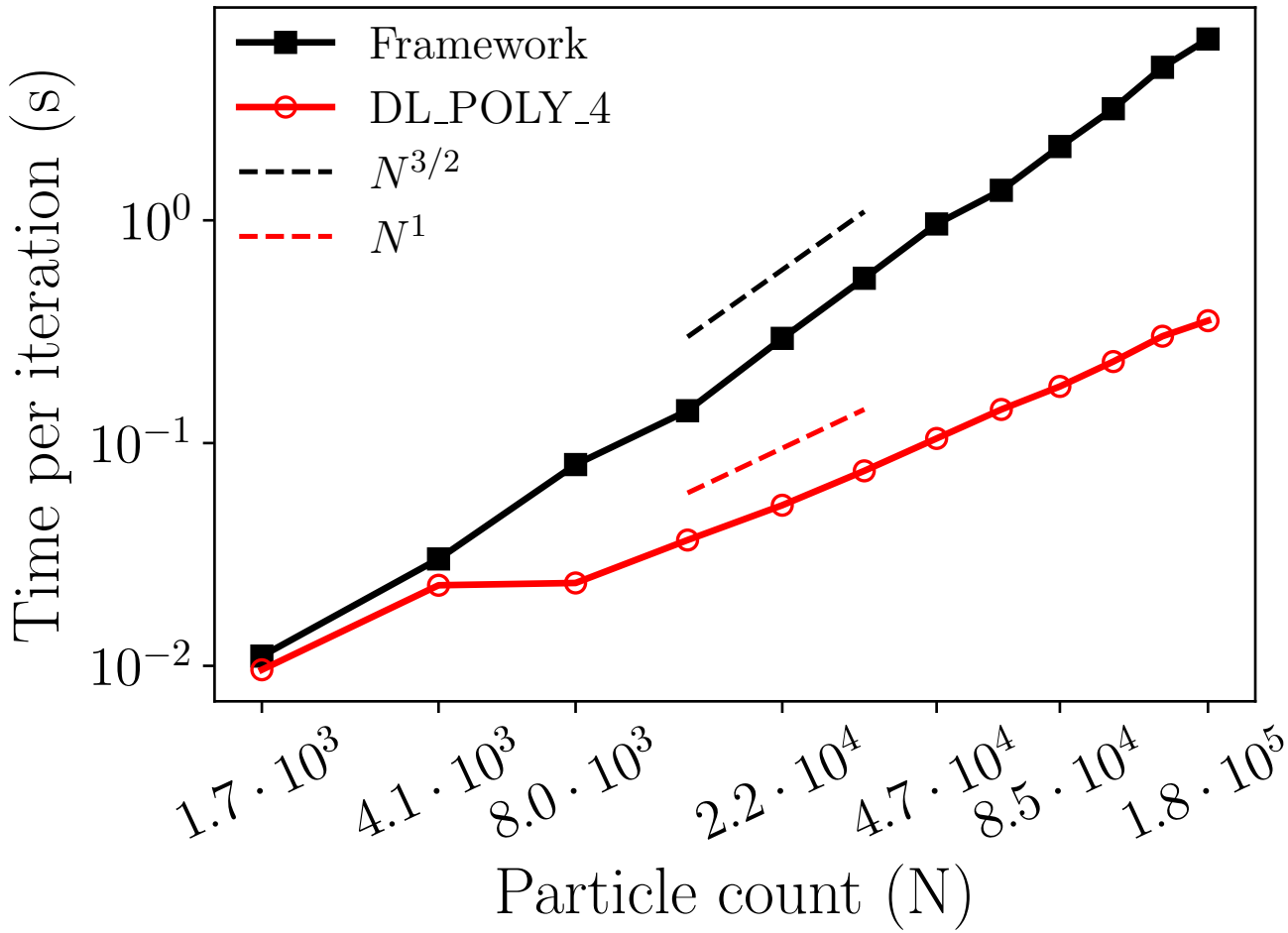
- $\rho^{(sr)}$: exponentially screened $\phi^{(sr)} \sim e^{-\alpha^2 r^2}$
  $\Rightarrow$ sum directly, truncate at $r_c \sim 1/\alpha \Rightarrow \text{Cost}^{(sr)} = C^{(sr)}(r_c) \cdot N$
- $\rho^{(lr)}$: Calculate in Fourier space, truncate at $k_c \sim \alpha$
  $\Rightarrow \text{Cost}^{(lr)} = C^{(lr)}(k_c) \cdot N$

Tune $\alpha(N)$, $r_c(\alpha)$ and $k_c(\alpha)$ to minimise total cost at fixed error [Kolafa and Perram (1992)]

$\Rightarrow$ **Computational complexity $\mathcal{O}(N^{3/2})$**

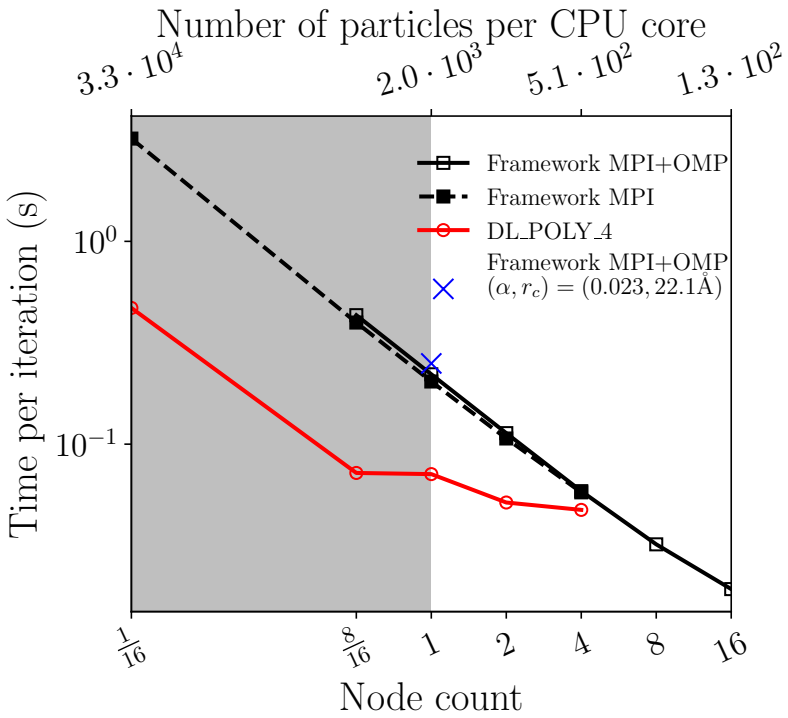
Results

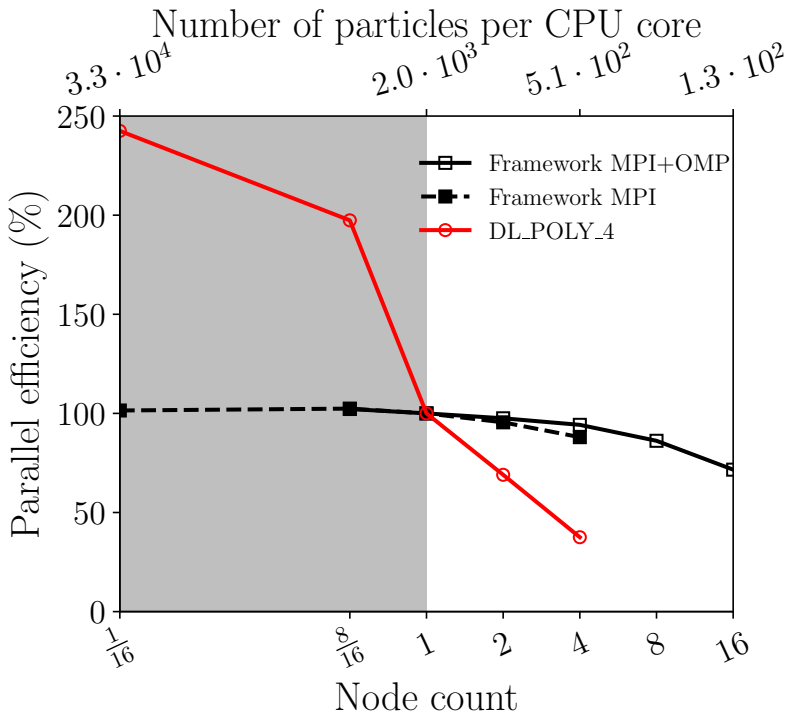
Computational Complexity [Saunders, Grant, Müller, arXiv:1708.01135]



Results

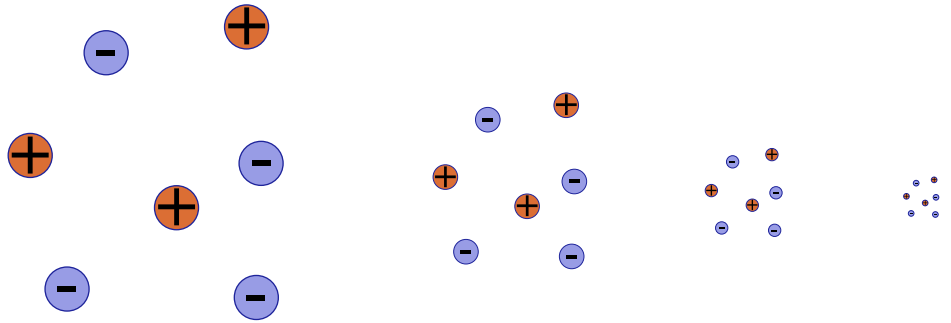
Parallel Scalability





Fast Multipole Method

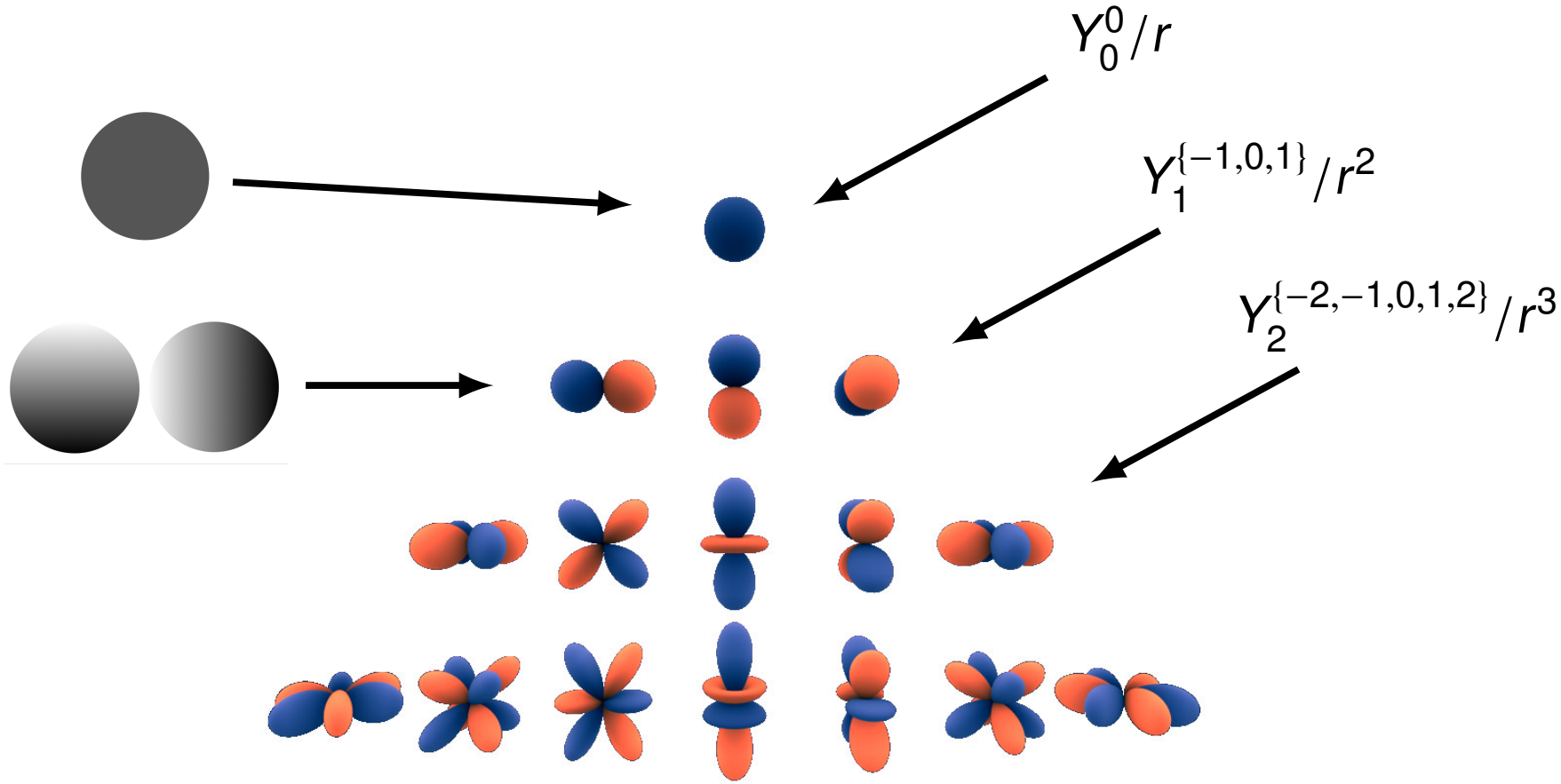
The exact structure of a cluster of charges becomes less important when observed from further distances.



Similar effect with images

Multipole expansion

Expansion in **Spherical Harmonics**



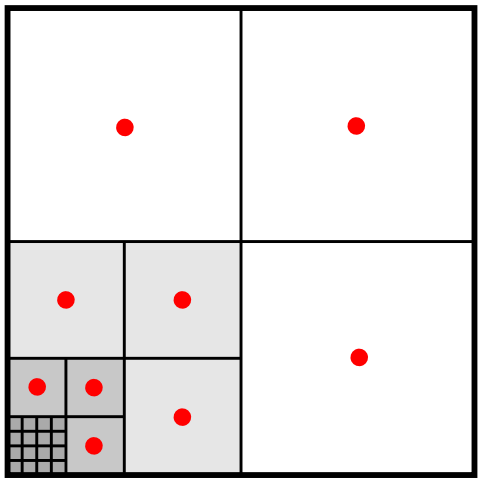
https://upload.wikimedia.org/wikipedia/commons/6/62/Spherical_Harmonics.png

Fast Multipole Method

Fast Multipole Method [Greengard and Rokhlin (1987)]

$$\phi(r,\theta,\phi) = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{+\ell} a_{\ell m} \frac{Y_{\ell,m}(\theta,\phi)}{r^{\ell+1}} \quad r > R$$

$$= \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{+\ell} b_{\ell m} Y_{\ell,m}(\theta,\phi) r^{\ell} \quad r \leq R$$



Upward pass

Multipole expansion on mesh hierarchy

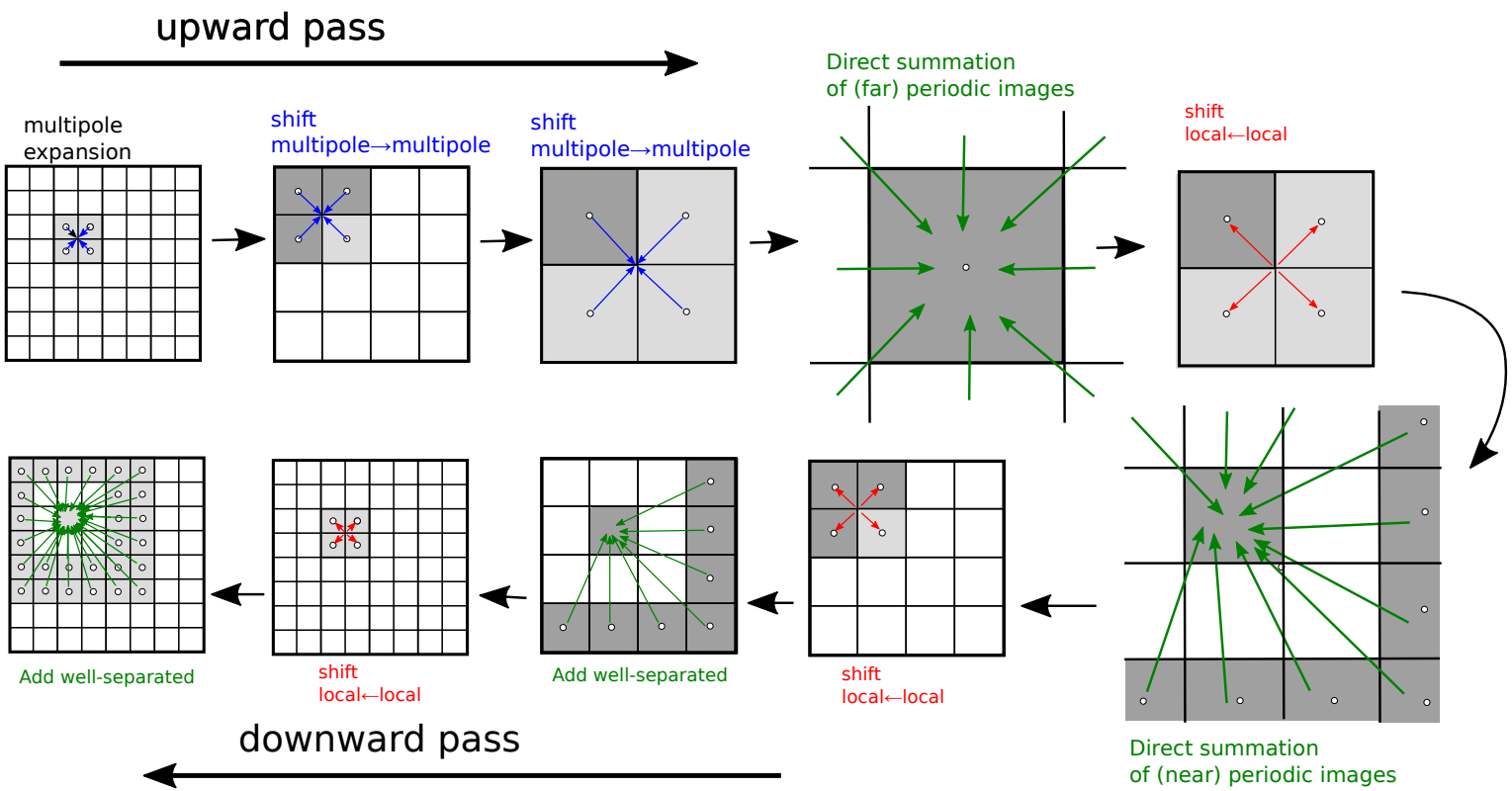
Downward pass

Increment local expansion

Evaluation of local expansion

$\Rightarrow$ **Computational Complexity** $\mathcal{O}(N)$

Fast Multipole Method



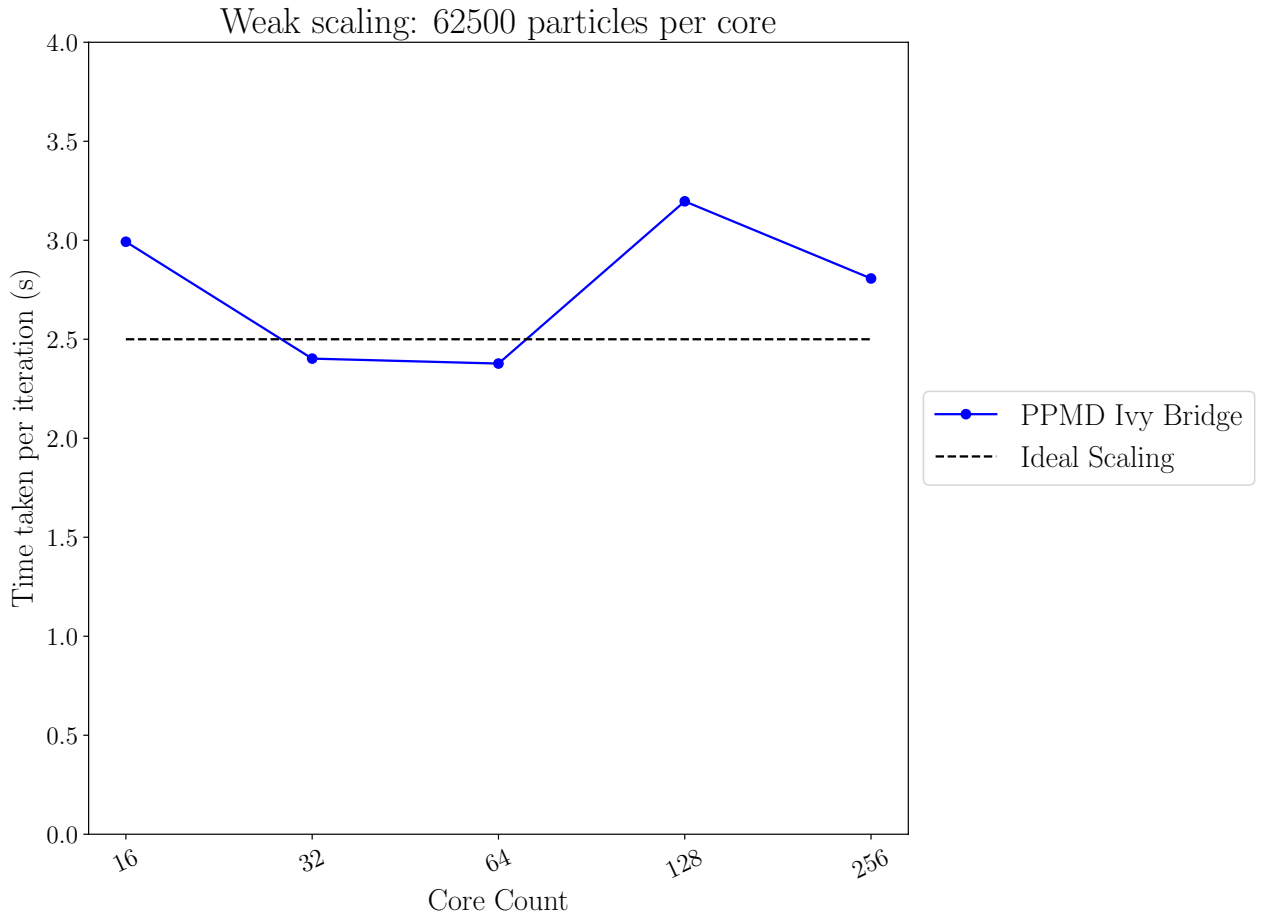
Setup

Setup

- DL_POLY NaCl benchmark
- electrostatics + repulsive LJ interactions (to prevent collapse)
- Charges arranged in a simple cubic lattice $a = 3.3\text{Å}$, random initial velocities
- periodic boundary conditions
- Constant density $\rho = (3.3\text{Å})^{-3}$
- Running 200 iterations, Velocity Verlet (no thermostat)
- particles wiggle around their initial positions
- Accuracy: error on total potential energy $\delta E/E \leq 10^{-5}$

Results

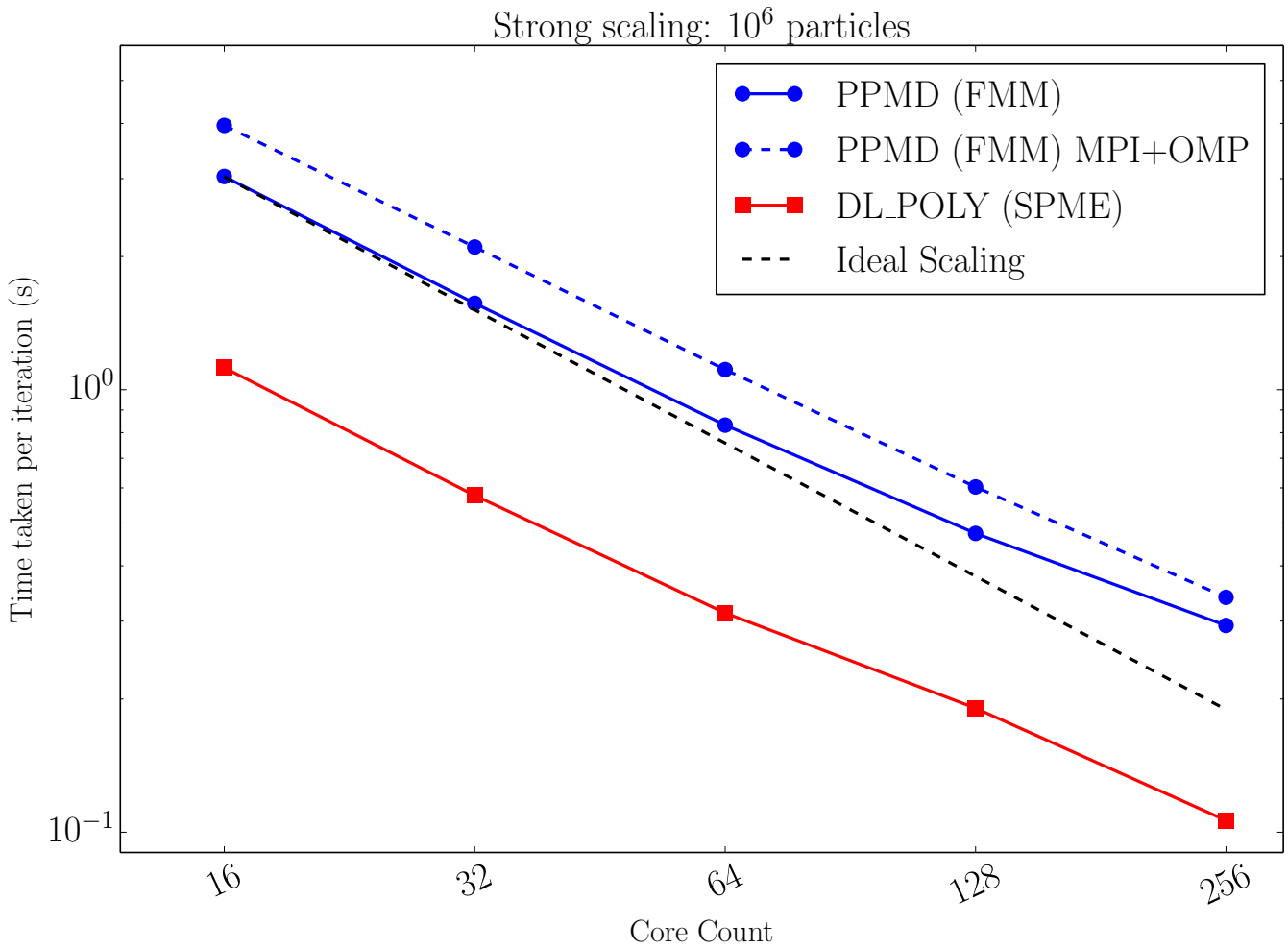
Weak scaling (1mio → 16mio particles, 16 → 256 cores)



⇒ Confirms $O(N)$ computational complexity

Results

Strong scaling $(62,500 \rightarrow 3,906$ particles/core, $16 \rightarrow 256$ cores)



Conclusion

Summary . . .

Saunders, Grant, Mueller (2018) CPC vol 224, pp. 119-135 and arXiv:1708.01135

- Performance portable (MPI, CUDA, CUDA+MPI, . . .)
- **Code generation** framework based on **“Separation of Concerns”**
- Speed/scalability comparable to established MD codes
- Arbitrary pair kernels, not just force calculation
- **Long range** electrostatic interactions (Ewald, Fast Multipole, . . .) currently CPU-only

. . . and Outlook

- GPU offload of FMM (in progress)
- Multiple species
- Constraints (molecules)