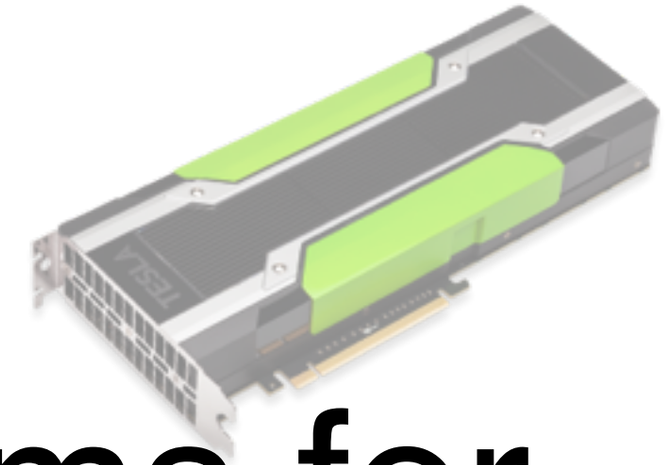
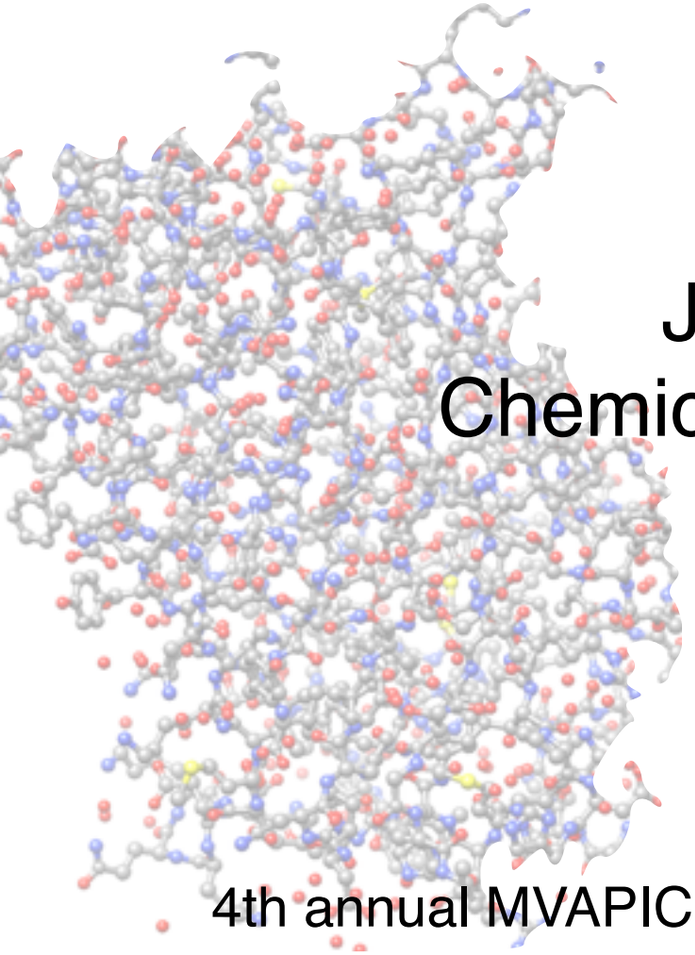


Distributed Algorithms for GPU Molecular Dynamics

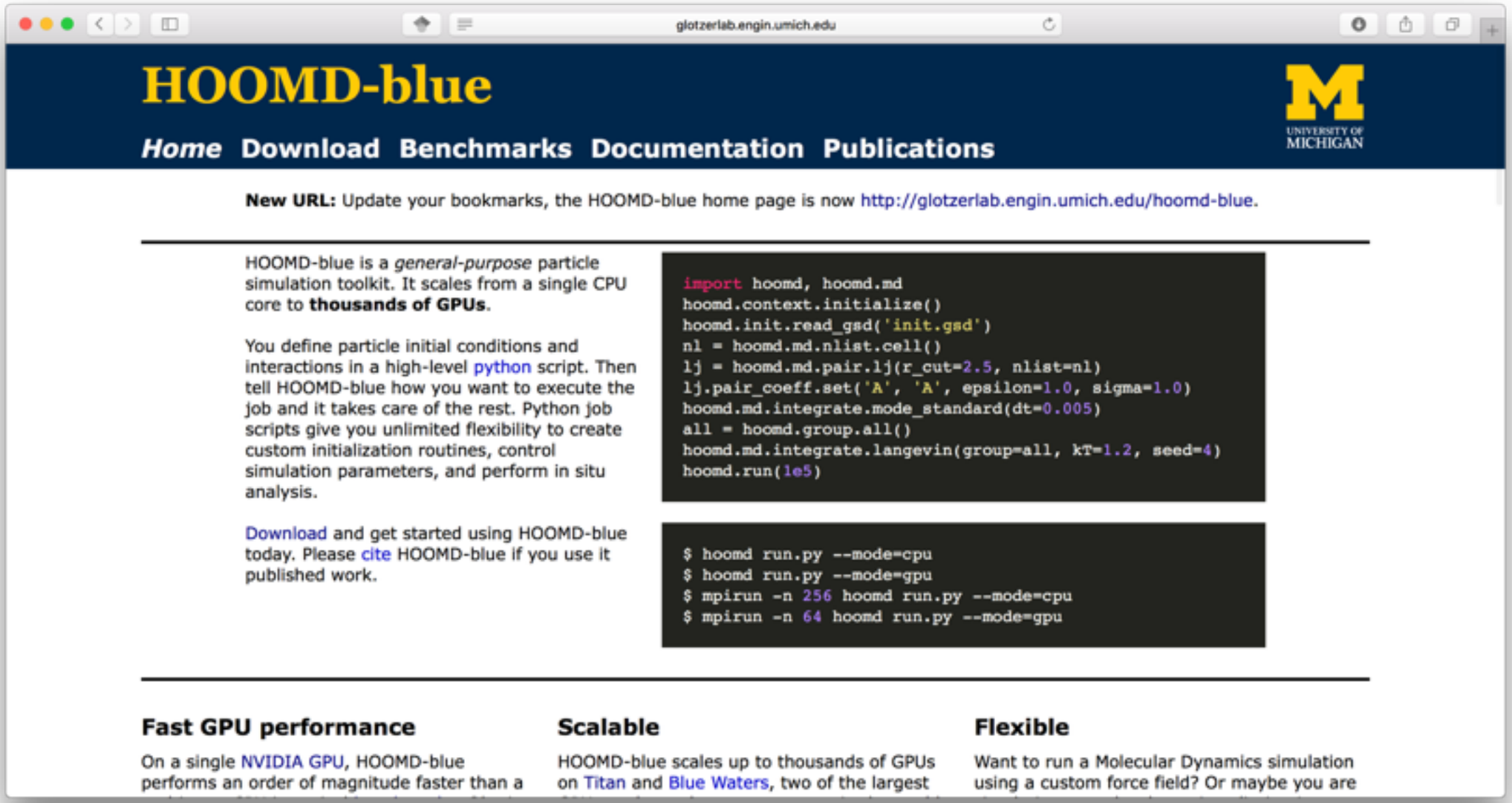


Jens Glaser, Postdoctoral Fellow
Chemical Engineering, University of Michigan

4th annual MVAPICH User Group (MUG) Meeting, Columbus, Ohio, August 15-17, 2016

HOOMD-blue

Highly **O**ptimized **O**bject-oriented **M**any **P**article **D**ynamics
Lead-developed in the Glotzer group at the University of Michigan
over 130 publications using HOOMD-blue since 2008



Webpage: http://glotzerlab.engin.umich.edu/hoomd-blue

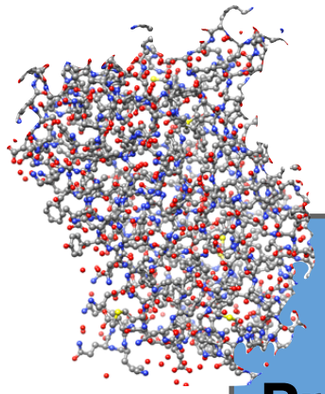
Documentation: http://hoomd-blue.readthedocs.io/en/stable/

Development and complete change log: https://bitbucket.org/glotzer/hoomd-blue

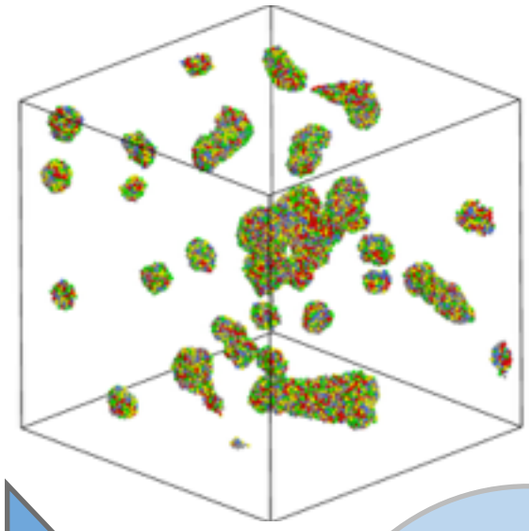
HOOMD-blue 2.0

- **New packages**
  - **hpmc** (Hard Particle Monte Carlo)
  - **dem** (Discrete Element Method)
- **New features**:
  - **md.constrain.distance**() - Pairwise distance constraints
  - **md.constrain.rigid**() - composite particles now have central particles and are supported with MPI
  - MPI support for **md.charge.pppm**() - distributed Coulomb force computation
  - **context.initialize()** can now be called multiple times, useful for Jupyter notebooks
  - Manage multiple concurrent simulation contexts in a single job script with **SimulationContext**
  - New binary GSD file format for storing trajectories, **dump.gsd(), init.gsd(), data.gsd_snapshot()**
  - **init.create_lattice()** to initialize from regular lattices
  - …

Computational biomaterial design



Protein structure (PDB)



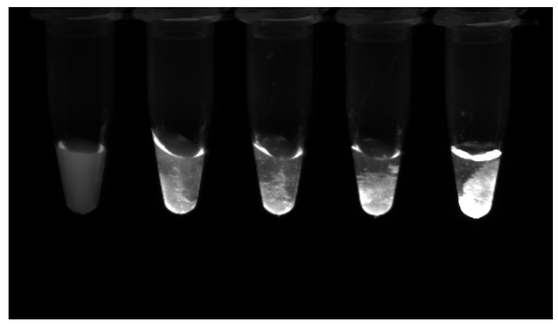
Self-assembly

Functional biomaterial

coarse-grained models

GPU Molecular Dynamics

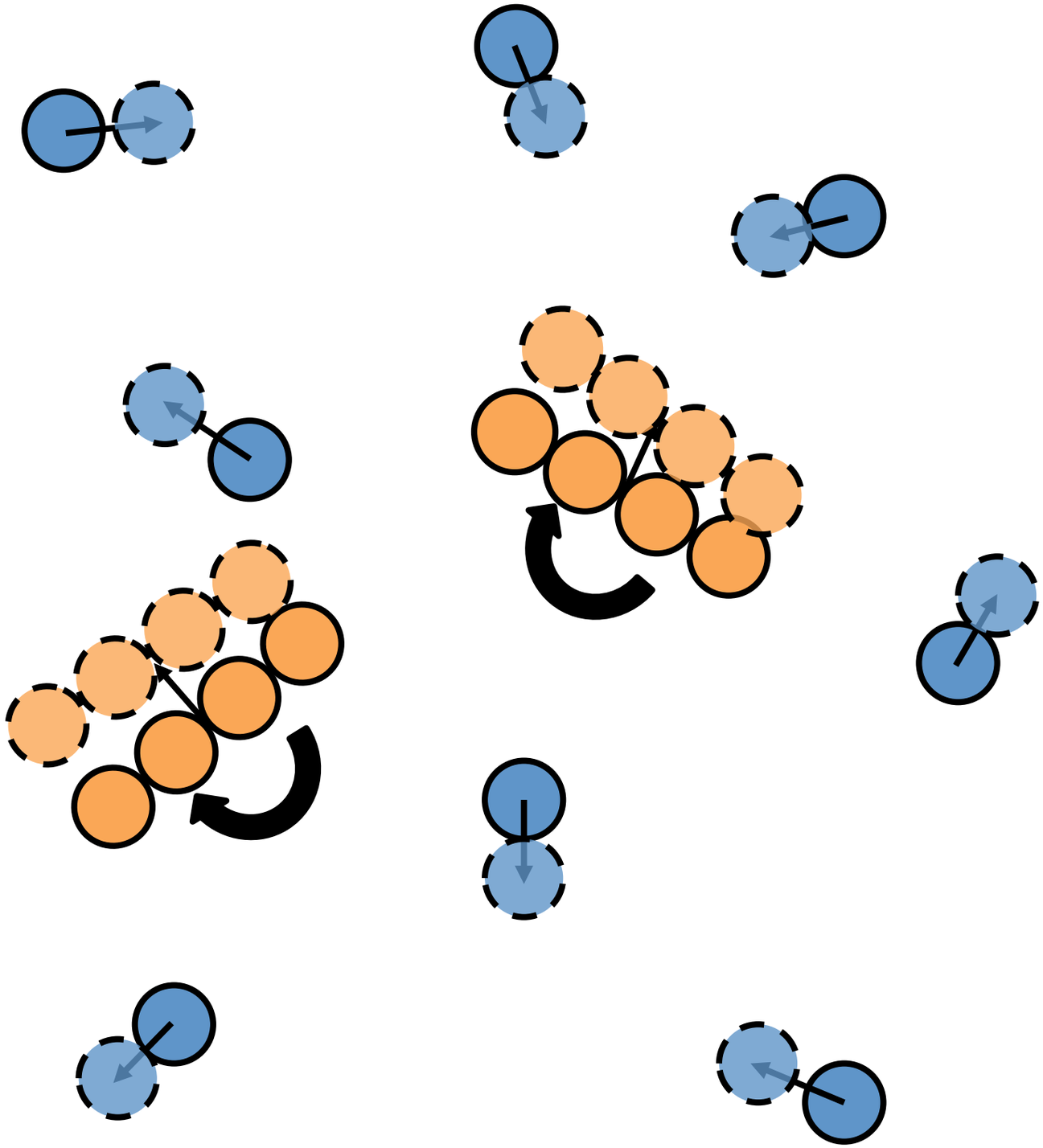
Experimental validation



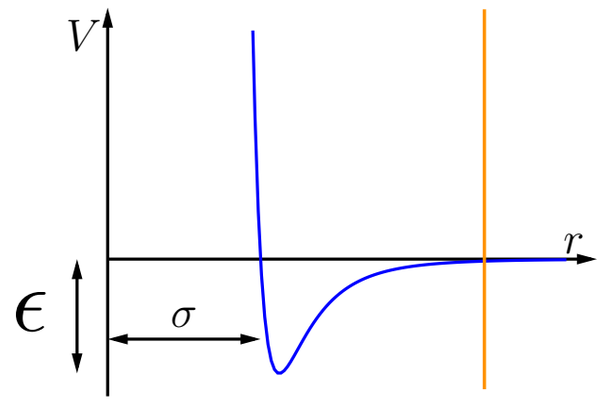
e.g. detection of a biothreat agent

Glotzer / Ellington collaboration

Molecular Dynamics



$$\vec{r}_i(t) \quad \vec{v}_i(t)$$
$$\vec{q}_i(t) \quad \vec{\omega}_i(t)$$

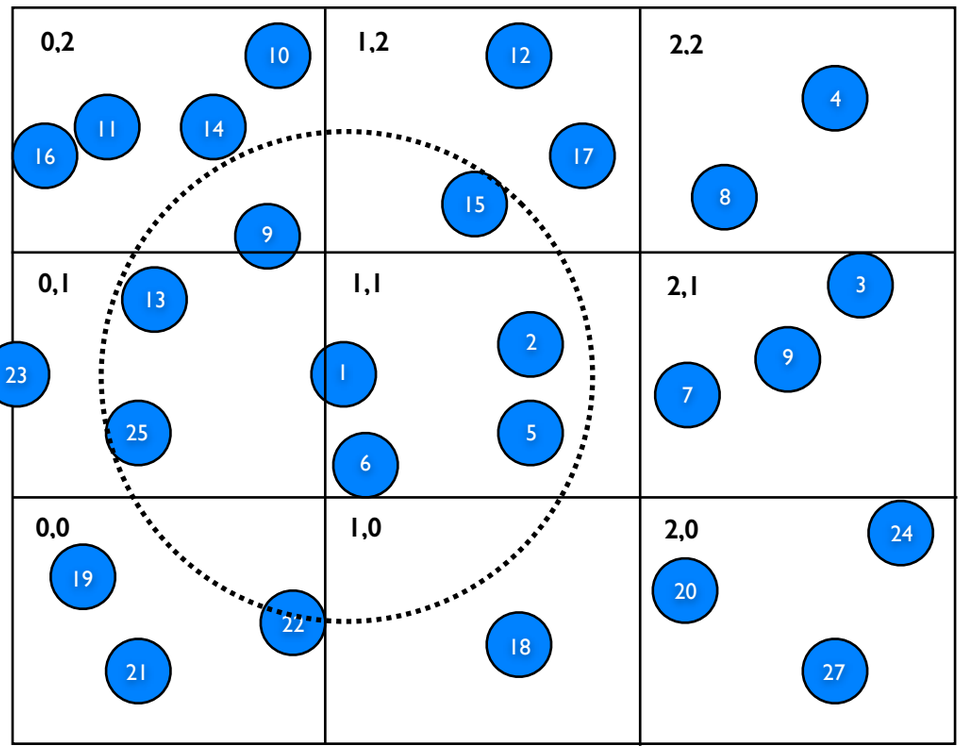


compute interactions

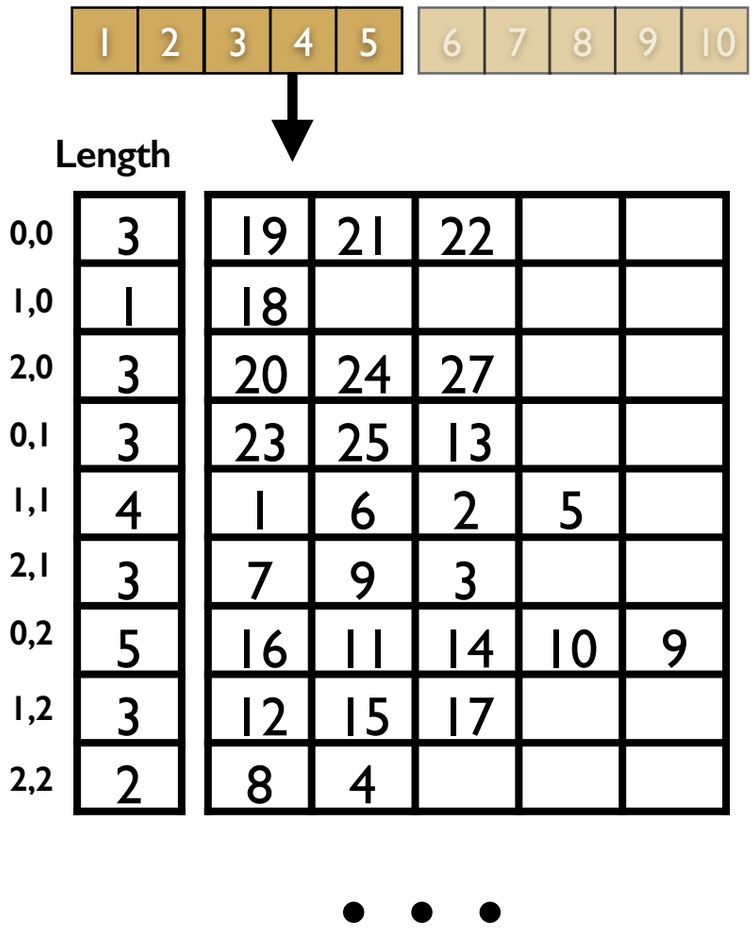
$$\vec{r}_i(t+\delta t) \quad \vec{v}_i(t+\delta t)$$
$$\vec{q}_i(t+\delta t) \quad \vec{\omega}_i(t+\delta t)$$

illustration by Joshua Anderson

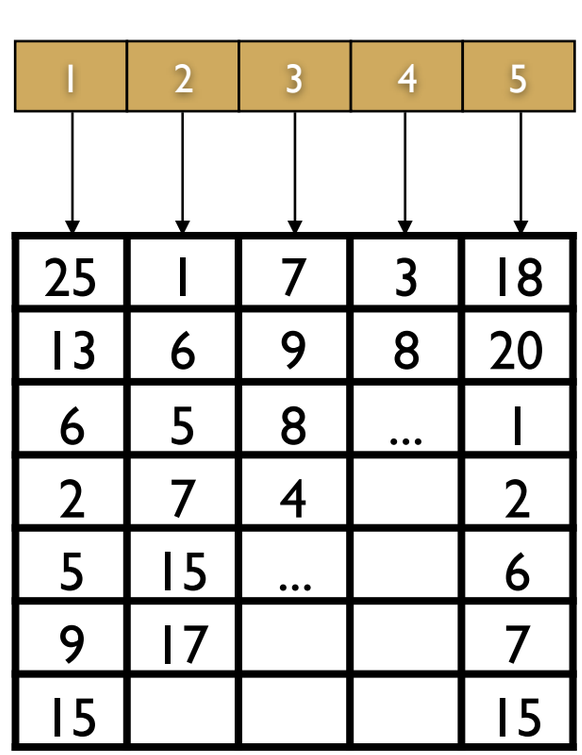
GPU Molecular Dynamics in 1 slide



Cell list



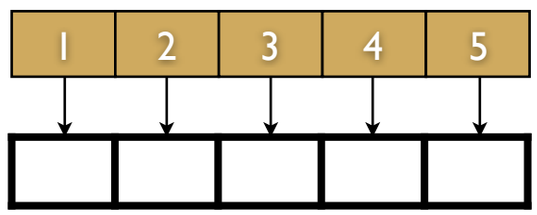
Neighbor list



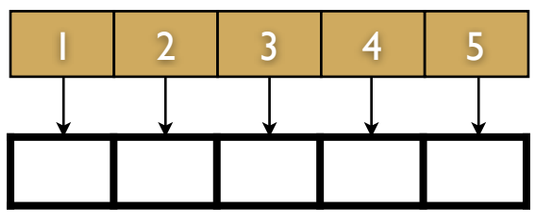
Neighbor list

Num Neighbors

Pair force

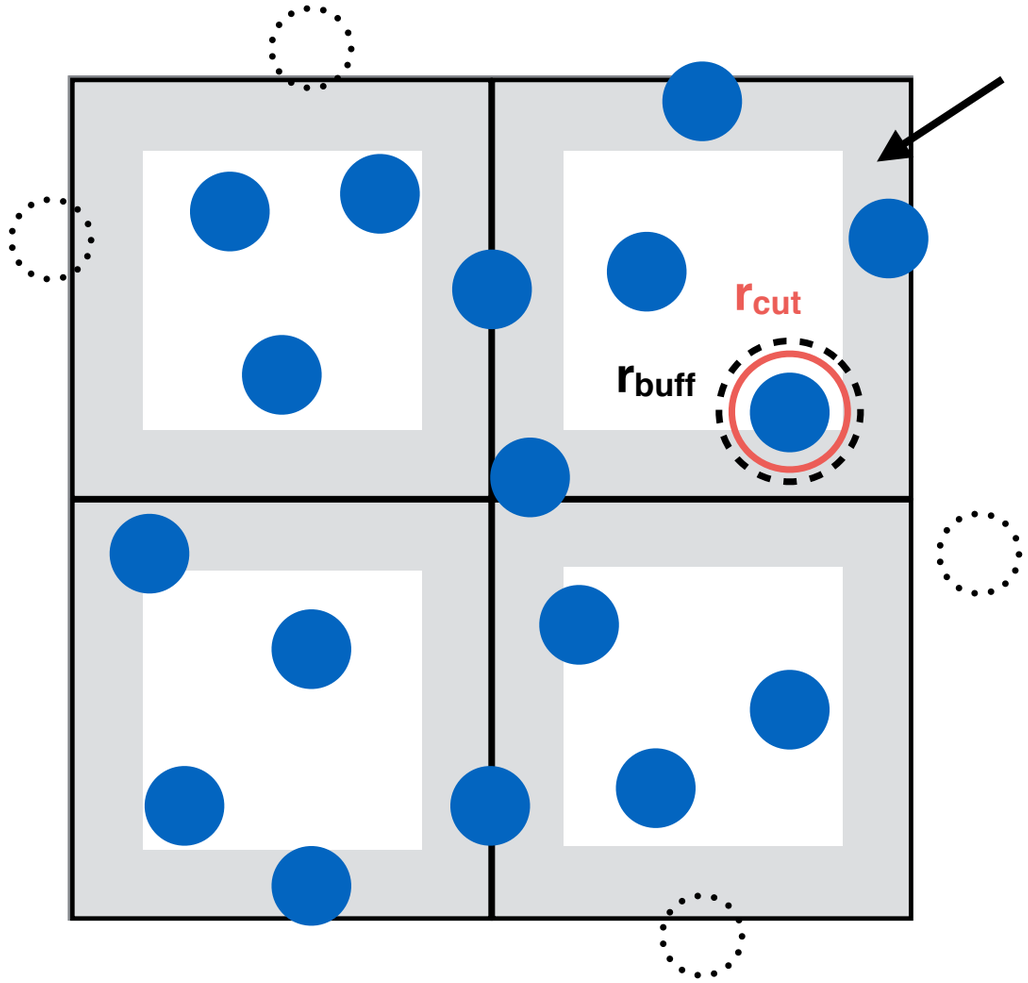


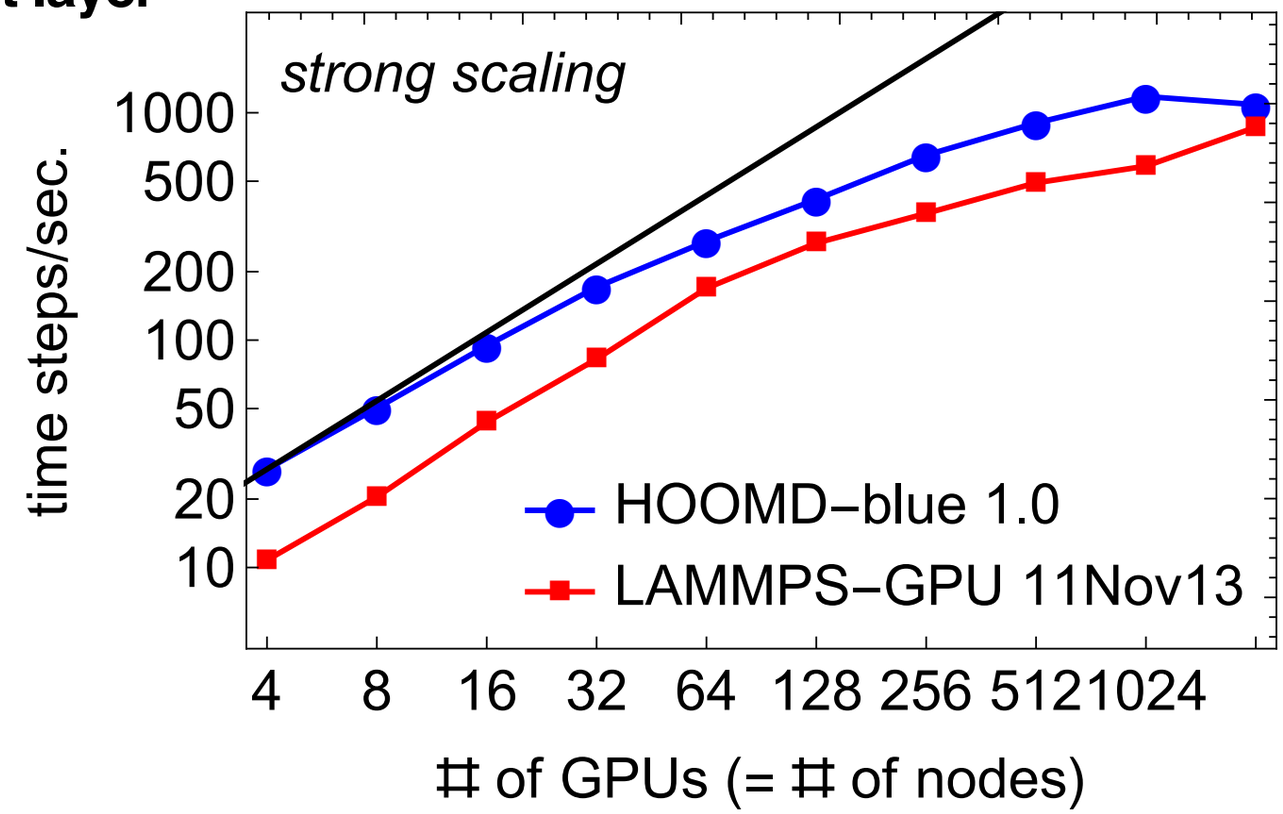
Integrate



slide by Joshua Anderson

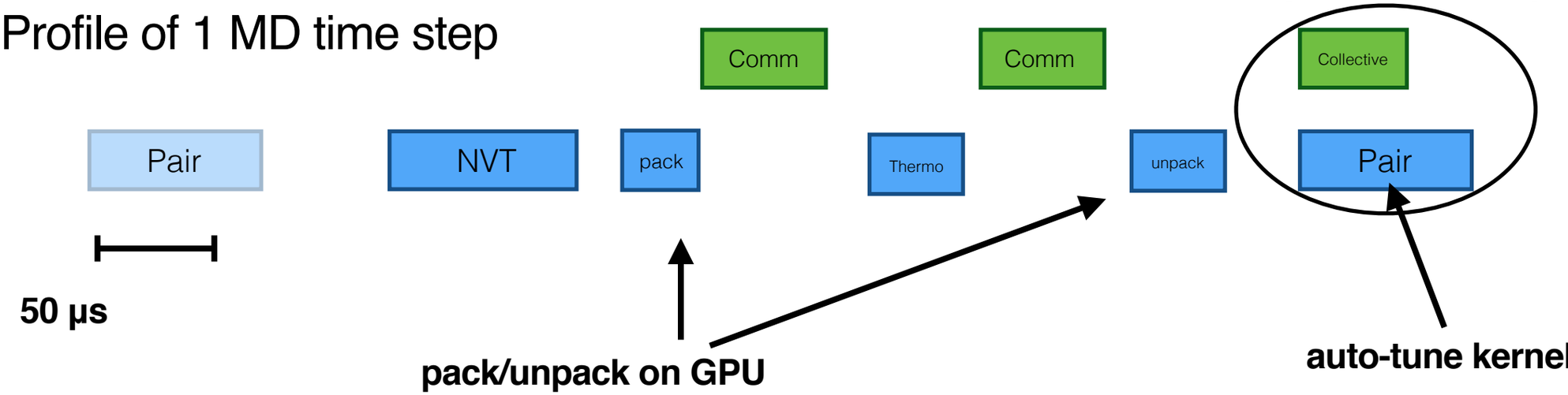
Domain decomposition with MPI





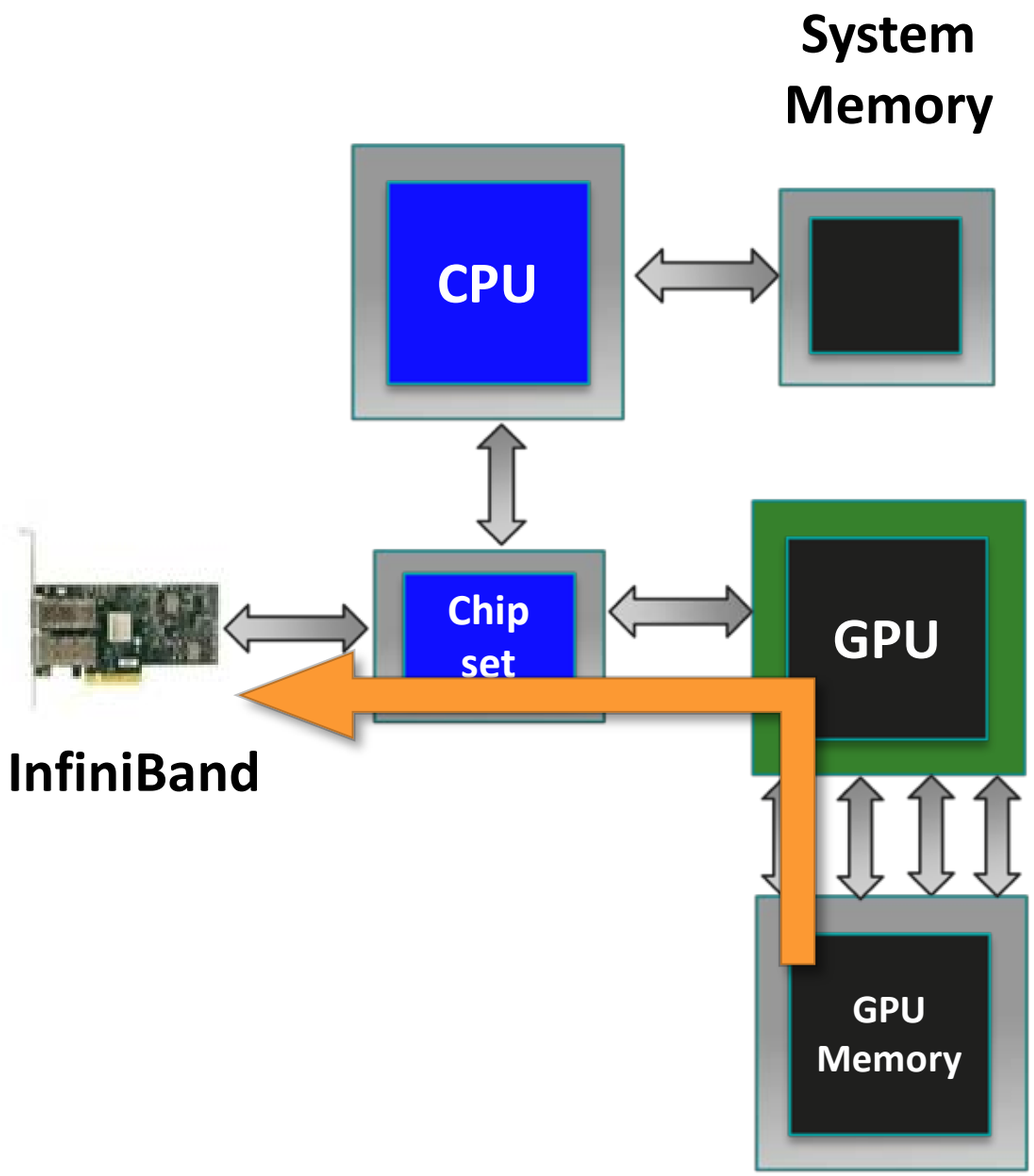
MPI Profile of 1 MD time step

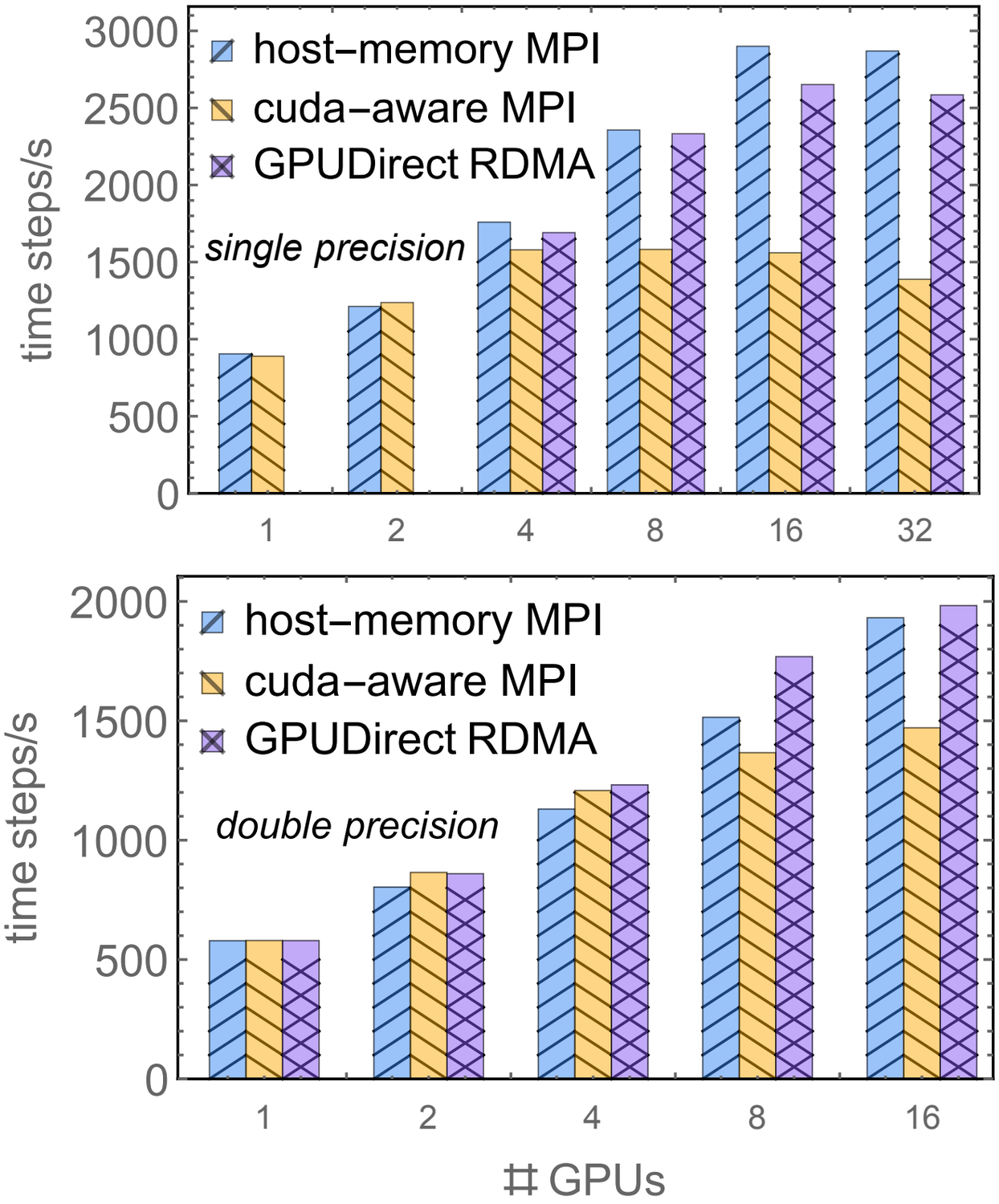
GPU



J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer, “Strong scaling of general-purpose molecular dynamics simulations on GPUs,” Comput. Phys. Commun., vol. 192, no. July, pp. 97–107, 2015.

GPUDirect RDMA





MVAPICH2 2.0 GDR data
courtesy of D.K. Panda and Rong Shi, OSU

Job script example — Lennard-Jones liquid

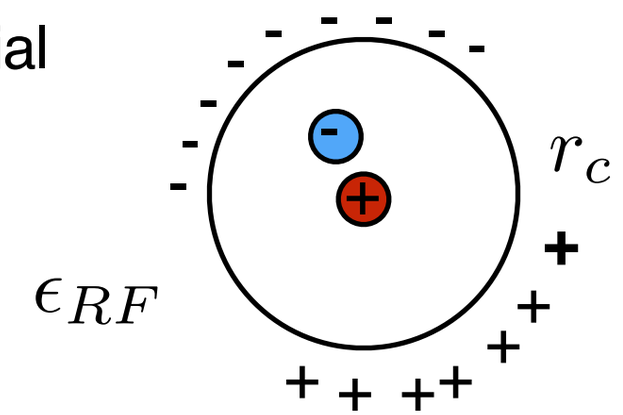
```
import hoomd, hoomd.md
hoomd.context.initialize()
hoomd.init.read_gsd('init.gsd')
nl = hoomd.md.nlist.cell()
lj = hoomd.md.pair.lj(r_cut=2.5, nlist=nl)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
hoomd.md.integrate.mode_standard(dt=0.005)
all = hoomd.group.all()
hoomd.md.integrate.langevin(group=all, kT=1.2, seed=4)
hoomd.run(1e5)
```

Charged interactions

Onsager reaction field

Polarization of the cut-off sphere due to 1/r potential

$$U(r) = q_A q_B \left[\frac{1}{r} + \frac{(\epsilon_{RF} - 1) r^2}{(2\epsilon_{RF} + 1) r_c^3} \right]$$



Ewald summation to compute long-range 1/r

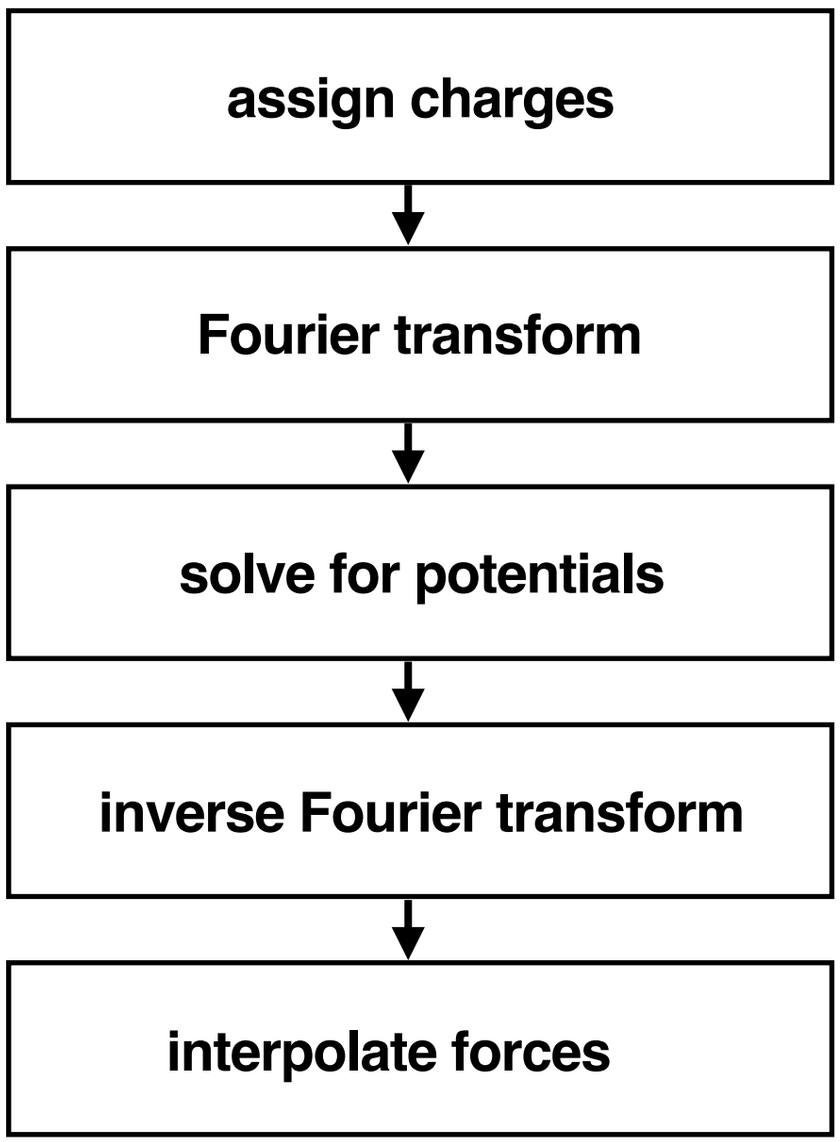
$$\frac{1}{r} = \frac{f(r)}{r} + \frac{1 - f(r)}{r} \qquad U(r_{ij}) = \frac{q_i q_j}{4\pi\varepsilon_0\varepsilon_r r_{ij}}$$

$$U_{\text{real}}(r_{ij}) = \frac{1}{2} \sum_{i,j} \frac{q_i q_j \operatorname{erfc}(\alpha r_{ij})}{4\pi\varepsilon_0\varepsilon_r r_{ij}} \qquad U_{\text{Fourier}} = \frac{1}{2} \sum_{k \neq 0} \frac{|\rho(\mathbf{k})|^2}{\varepsilon_0\varepsilon_r V k^2} \exp(-k^2/4\alpha^2) - \frac{\alpha}{\sqrt{(\pi)}} \sum_i q_i^2,$$

D. N. LeBard, B. G. Levine, P. Mertmann, S. A. Barr, A. Jusufi, S. Sanders, M. L. Klein, and A. Z. Panagiotopoulos
Self-assembly of coarse-grained ionic surfactants accelerated by graphics processing units
Soft Matter, vol. 8, no. 8, pp. 2385–2397, 2012.

Particle-particle Particle-Mesh (PPPM) electrostatics

PM force computation



$$\rho_g = \frac{1}{h}\sum_{i=1}^{N} q_i W(\mathbf{r}_p - \mathbf{r}_i), \qquad W_P(s) = \underbrace{\left(\mathbf{1}_{[-1/2,1/2]} \bigstar \cdots \bigstar \mathbf{1}_{[-1/2,1/2]}\right)}_{p-\text{fold convolution}}(x/h)$$

$$\hat{\rho}_{\mathbf{k_n}} = \frac{1}{V}\sum_{\mathbf{r_n}} e^{i\mathbf{k_n}\cdot\mathbf{r_n}} q_{\mathbf{r_n}},$$

$$\hat{\phi}_{\mathbf{k_n}} = \hat{G}'_{\mathbf{k_n}}\hat{\rho}_{\mathbf{k_n}}. \qquad \hat{G}'_{\mathbf{k_n}} = \frac{\sum_{\mathbf{M}} \hat{U}^2_{\mathbf{k_n}+\mathbf{M}}\hat{G}_{\mathbf{k_n}+\mathbf{M}}\mathbf{k_n}\cdot\mathbf{k_{n+M}}}{\left(\sum_{\mathbf{M}} \hat{U}^2_{\mathbf{k_n}+\mathbf{M}}\right)^2 |\mathbf{k_n}|^2} \qquad \hat{G}_{\mathbf{k}} \equiv \frac{4\pi}{|\mathbf{k}|^2} e^{-|\mathbf{k}|^2/4\alpha^2}$$

$$\hat{U}_{\mathbf{k}} \equiv \frac{N_X N_Y N_Z}{V}\hat{W}_{\mathbf{k}},$$

$$\mathbf{E}(\mathbf{r_p}) = -\overleftarrow{\text{FFT}}\left[i\mathbf{k}\times\hat{\rho}_{\mathbf{k_n}}\times\hat{G}'_{\mathbf{k_n}}\right](\mathbf{r_p}).$$

$$\mathbf{F}_i = q_i \sum_{\mathbf{r}_p\in\mathbf{M}} \mathbf{E}(\mathbf{r}_p) W(\mathbf{r}_p - \mathbf{r}_i)$$

Fast Fourier Transform

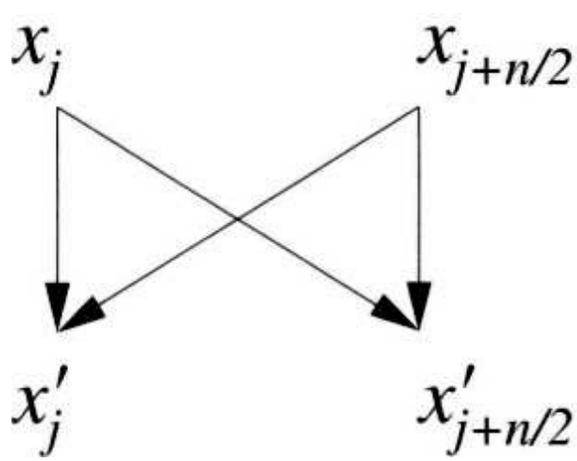
$$y_k = \sum_{j=0}^{n-1} x_j e^{-2\pi ijk/n} \quad 0 \le k < n \qquad n = 2^m$$

$$y_k = \sum_{j=0}^{n-1} x_{2j}\omega_n^{2jk} + \sum_{j=0}^{n/2-1} x_{2j+1}\omega_n^{(2j+1)k} \qquad \omega_n = e^{-2\pi i/n}$$

$$\omega_n^2 = \omega_{n/2}$$

$$y_k = \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{jk} + \omega_n^k \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{jk}$$

$$y_{k+n/2} = \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{jk} - \omega_n^k \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{jk}$$



R. H. Bisseling, Parallel Scientific Computation. Oxford University Press, 2004.

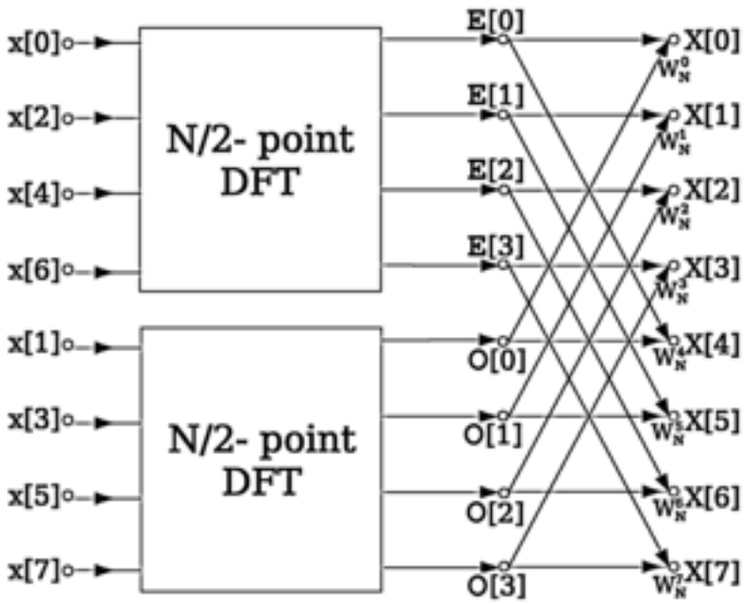
Recursive O[N log(N)] algorithm

input: $\mathbf{x}$: vector of length n.
output: $\mathbf{y}$: vector of length n. $\mathbf{y} = F_n \mathbf{x}$.
function call: $\mathbf{y} := \mathrm{FFT}(\mathbf{x}, n)$.

if $n \bmod 2 = 0$ **then**
$\mathbf{x}^{\mathrm{e}} := x(0\colon 2\colon n-1)$;
$\mathbf{x}^{\mathrm{o}} := x(1\colon 2\colon n-1)$;
$\mathbf{y}^{\mathrm{e}} := \mathrm{FFT}(\mathbf{x}^{\mathrm{e}}, n/2)$;
$\mathbf{y}^{\mathrm{o}} := \mathrm{FFT}(\mathbf{x}^{\mathrm{o}}, n/2)$;
for $k := 0$ **to** $n/2 - 1$ **do**
$\tau := \omega_n^k y_k^{\mathrm{o}}$;
$y_k := y_k^{\mathrm{e}} + \tau$;
$y_{k+n/2} := y_k^{\mathrm{e}} - \tau$;
else $\mathbf{y} := \mathrm{DFT}(\mathbf{x}, n)$;

R. H. Bisseling, Parallel Scientific Computation. Oxford University Press, 2004.

Non-recursive FFT with bit reversal



LSB ⇔ MSB

https://en.wikipedia.org/wiki/Cooley–Tukey_FFT_algorithm

input: $\mathbf{x}$: vector of length $n = 2^m$, $m \geq 1$, $\mathbf{x} = \mathbf{x}_0$.
output: $\mathbf{x}$: vector of length n, such that $\mathbf{x} = F_n \mathbf{x}_0$.
function call: FFT$(\mathbf{x}, n)$.

{ Perform bit reversal $\mathbf{x} := R_n \mathbf{x}$. Function call bitrev$(\mathbf{x}, n)$ }
for $j := 0$ **to** $n - 1$ **do**
    { Compute $r := \rho_n(j)$ }
    $q := j$;
    $r := 0$;
    for $k := 0$ **to** $\log_2 n - 1$ **do**
        $b_k := q \bmod 2$;
        $q := q$ div 2;
        $r := 2r + b_k$;
    if $j < r$ **then** swap(x_j, x_r);

{ Perform butterflies. Function call UFFT$(\mathbf{x}, n)$ }
$k := 2$;
while $k \leq n$ **do**
    { Compute $\mathbf{x} := (I_{n/k} \otimes B_k)\mathbf{x}$ }
    for $r := 0$ **to** $\frac{n}{k} - 1$ **do**
        { Compute $x(rk : rk + k - 1) := B_k x(rk : rk + k - 1)$ }
        for $j := 0$ **to** $\frac{k}{2} - 1$ **do**
            { Compute $x_{rk+j} \pm \omega_k^j x_{rk+j+k/2}$}
            $\tau := \omega_k^j x_{rk+j+k/2}$;
            $x_{rk+j+k/2} := x_{rk+j} - \tau$;
            $x_{rk+j} := x_{rk+j} + \tau$;
    $k := 2k$;

R. H. Bisseling, Parallel Scientific Computation. Oxford University Press, 2004.

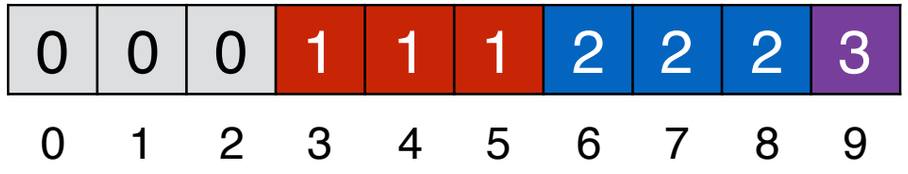
Distributed FFT

$x_i = x_0, x_1, \ldots, x_n$ input vector

Block distribution on p processors

$x_i \rightarrow P(i \operatorname{div} b) \qquad b = \lceil n/p \rceil$

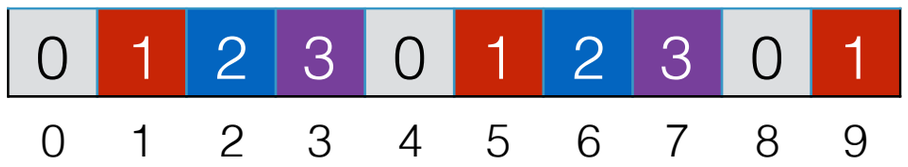
$n = 10$



Butterflies B_k with $k \leq n/p$ are local

Cyclic distribution

$x_i \rightarrow P(i \bmod p)$



Butterflies B_k with $k \geq 2p$ are local

↝ **Strategy: start with block distribution, finish with cyclic**

R. H. Bisseling, Parallel Scientific Computation. Oxford University Press, 2004.

Group-cyclic distribution

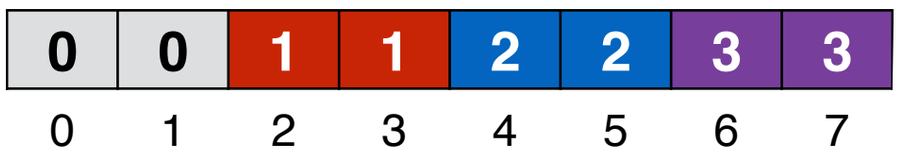
When too many processors are available, $p > n/p$, we need the

Group-Cyclic distribution

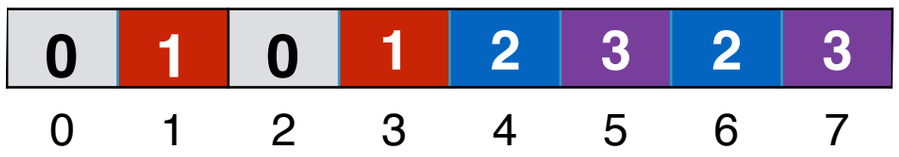
$$x_j \rightarrow P\left[\left(j \operatorname{div} \left\lceil \frac{cn}{p} \right\rceil\right) + \left(j \bmod \left\lceil \frac{cn}{p} \right\rceil\right) \bmod c\right]$$

$n = 8$

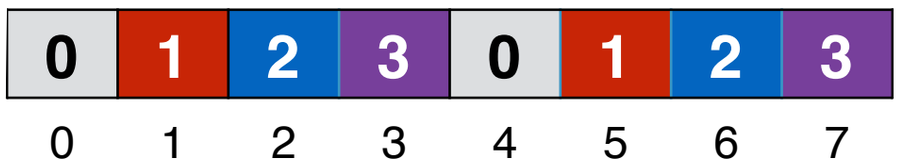
$c = 1$ **(block)**



$c = 2$



$c = 4$ **(cyclic)**



B_k with $2c \leq k \leq \frac{n}{p}c$

is local

R. H. Bisseling, Parallel Scientific Computation. Oxford University Press, 2004.

Algorithm - one-dimensional parallel FFT

input: $\mathbf{x}$: vector of length $n = 2^m$, $m \geq 1$, $\mathbf{x} = \mathbf{x}_0$,
distr($\mathbf{x}$) = cyclic over $p = 2^q$ processors with $0 \leq q < m$.
output: $\mathbf{x}$: vector of length n, distr($\mathbf{x}$) = cyclic, such that $\mathbf{x} = F_n \mathbf{x}_0$.

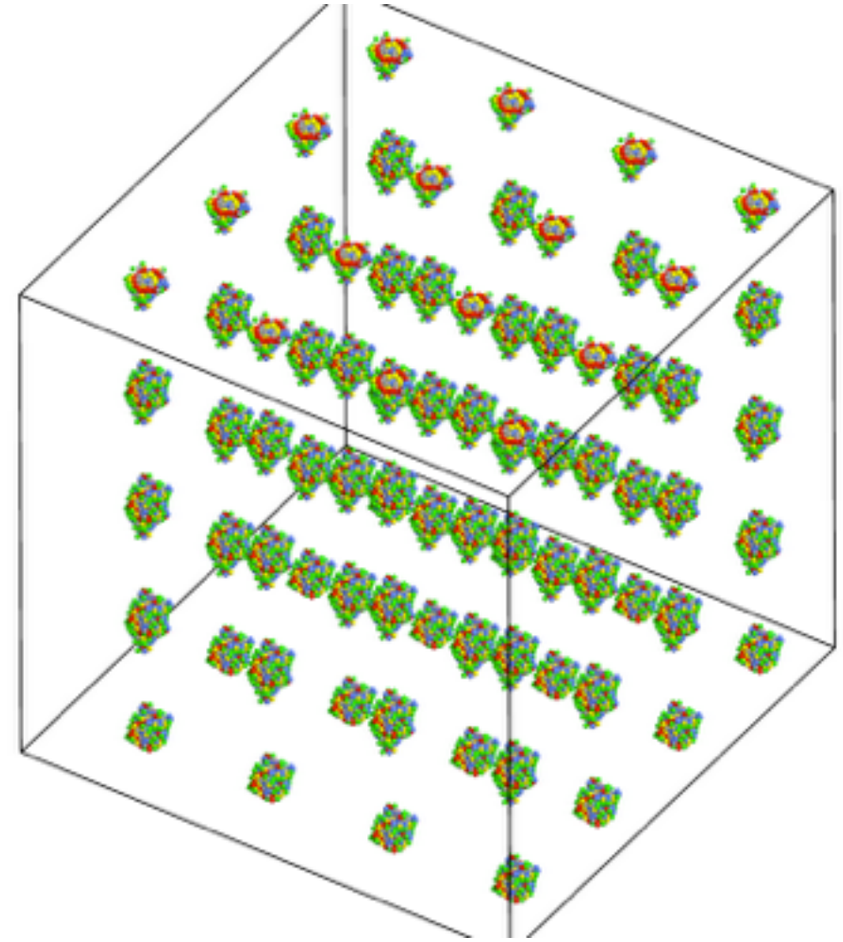
bitrev($x(s\colon p\colon n-1), n/p$);
{ distr($\mathbf{x}$) = block with bit-reversed processor numbering }

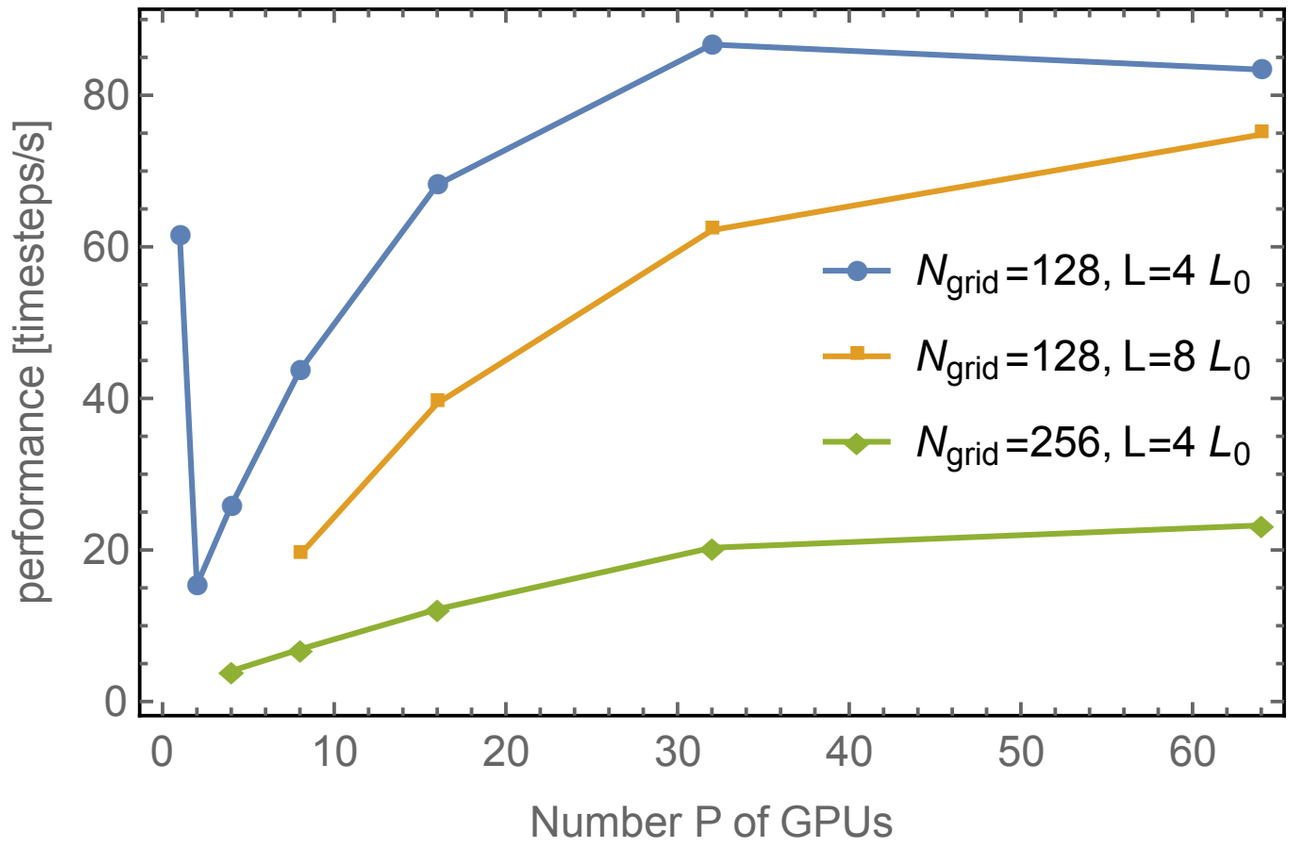
$k := 2$;
$c := 1$;
$rev := true$;
while $k \leq n$ **do**
(0) $j_0 := s \bmod c$;
$j_2 := s \operatorname{div} c$;
while $k \leq \frac{n}{p} c$ **do**
$nblocks := \frac{nc}{kp}$;
for $r := j_2 \cdot nblocks$ **to** $(j_2 + 1) \cdot nblocks - 1$ **do**
{ Compute local part of $x(rk\colon (r+1)k-1)$ }
for $j := j_0$ **to** $\frac{k}{2} - 1$ **step** c **do**
$\tau := \omega_k^j x_{rk+j+k/2}$;
$x_{rk+j+k/2} := x_{rk+j} - \tau$;
$x_{rk+j} := x_{rk+j} + \tau$;
$k := 2k$;
if $c < p$ **then**
$c_0 := c$;
$c := \min(\frac{n}{p} c, p)$;
(1) redistr($\mathbf{x}, n, p, c_0, c, rev$);
$rev := false$;
{ distr($\mathbf{x}$) = group-cyclic with cycle c }

R. H. Bisseling, Parallel Scientific Computation. Oxford University Press, 2004.

Implementation in HOOMD-blue 2.0 - md.charge.pppm()

- Local Butterflies are optimized using NVIDIA CUFFT on GPU
- Vector-Radix for n-d transforms
- Group-cyclic redistribution using MPI_Alltoallv()
- Implemented in dfftlib (http://github.com/jglaser/dfftlib)
- currently only single precision

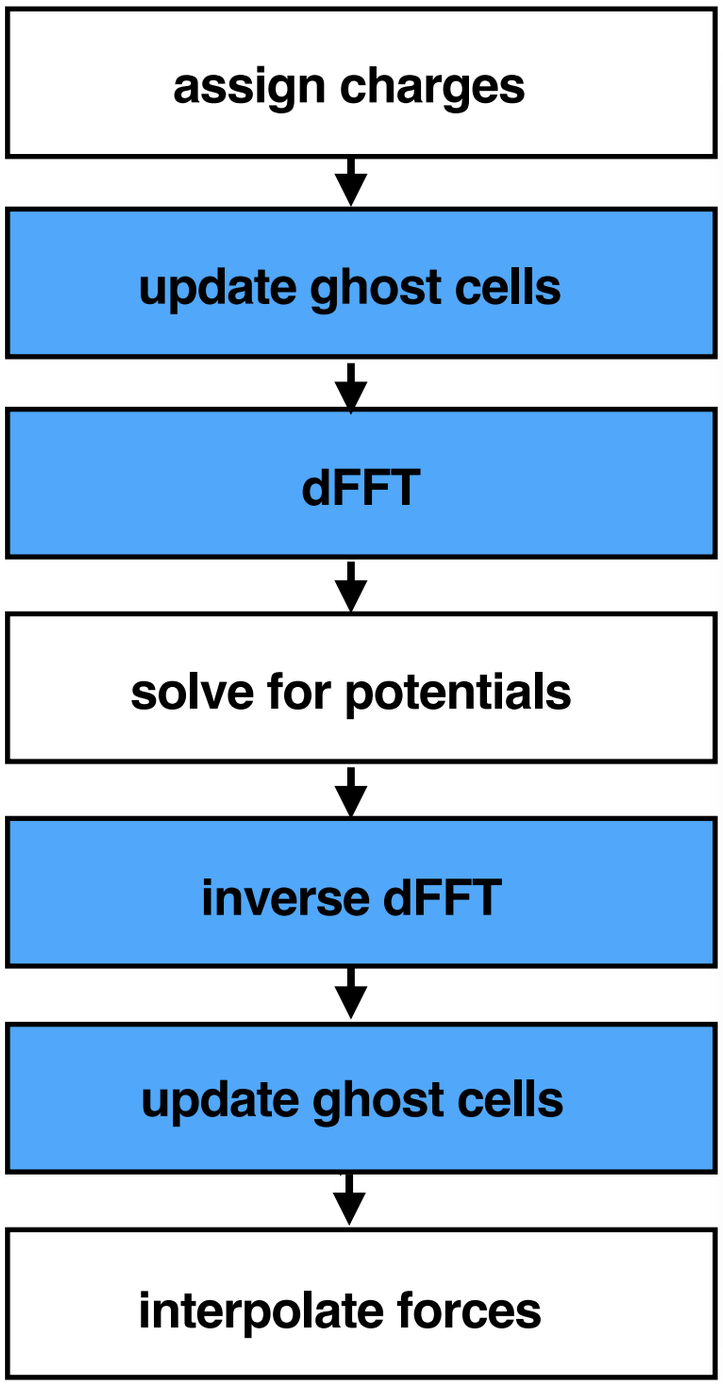


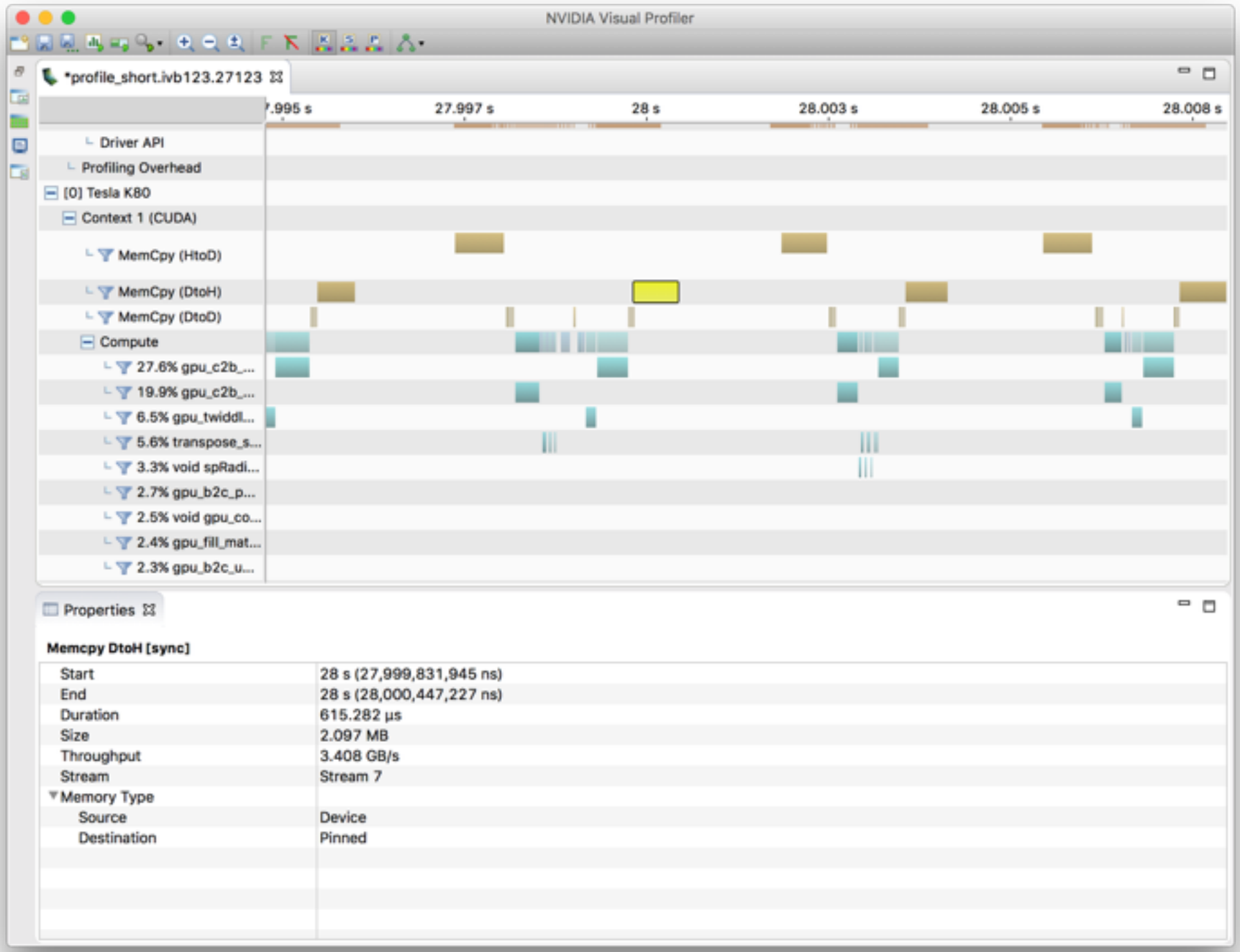


Protein aggregation benchmark
Martini FF w/long-range PPPM, no solvent

Ghost cell layer

PM force computation w/communication





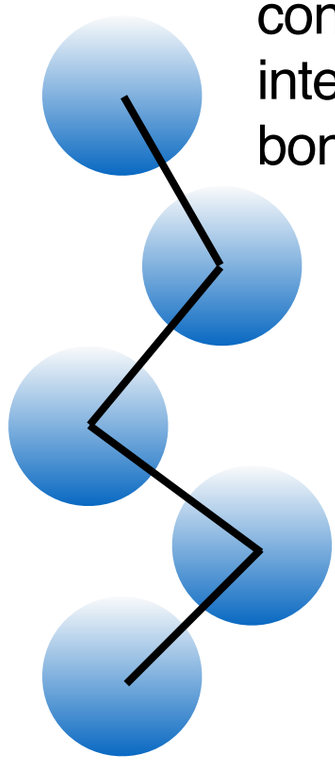
Pairwise distance constraints

Constraint equation $\chi_n(t) \equiv \mathbf{r}_n^2(t) - \mathbf{d}_n^2 = 0 \quad (n = 1 - N)$

$$\mathbf{r}_n(t) \equiv \mathbf{x}_i(t) - \mathbf{x}_j(t).$$

Constraint force $f_i^c(t) = \frac{1}{2}\sum_{n=1}^{N} \lambda_n(t) \nabla_{r_i(t)} \chi_n(t) = \sum_{n=1}^{N} \lambda_n(t) \boldsymbol{r}_i(t).$

Iterative SHAKE: $\chi_n(t + \Delta t) + O(\varepsilon) = 0.$



Distance constraints to integrate out fast bond fluctuations

Non-Iterative Matrix Method: $\chi_n(t + \Delta t) + O(\Delta t^4) = 0$

Linear matrix equation due to coupled constraints (combine with Velocity-Verlet):

$$\chi_n(t + 2\Delta t) + O(\Delta t^4) = \mathbf{q}_n^2(t + \Delta t) - \mathbf{d}_n^2 + 2\mathbf{q}_n(t + \Delta t) \cdot \ddot{\mathbf{r}}(t + \Delta t)\Delta t^2 = 0,$$

$$\mathbf{q}_n(t + \Delta t) \equiv \mathbf{r}_n(t + \Delta t) + \dot{\mathbf{r}}_n(t + \Delta t/2)\Delta t.$$

[1] M. Yoneya, H. J. C. Berendsen, and K. Hirasawa, “A Non-Iterative Matrix Method for Constraint Molecular Dynamics Simulations,” Mol. Simul., vol. 13, no. 6, pp. 395–405, 1994.
[2] M. Yoneya, “A Generalized Non-iterative Matrix Method for Constraint Molecular Dynamics Simulations,” J. Comput. Phys., vol. 172, no. 1, pp. 188–197, Sep. 2001.

Sparse matrix refactorization - md.constrain.distance()

At every time step, solve for the forces $\mathbf{M}\,\boldsymbol{\lambda} = v$

M: constraint topology matrix

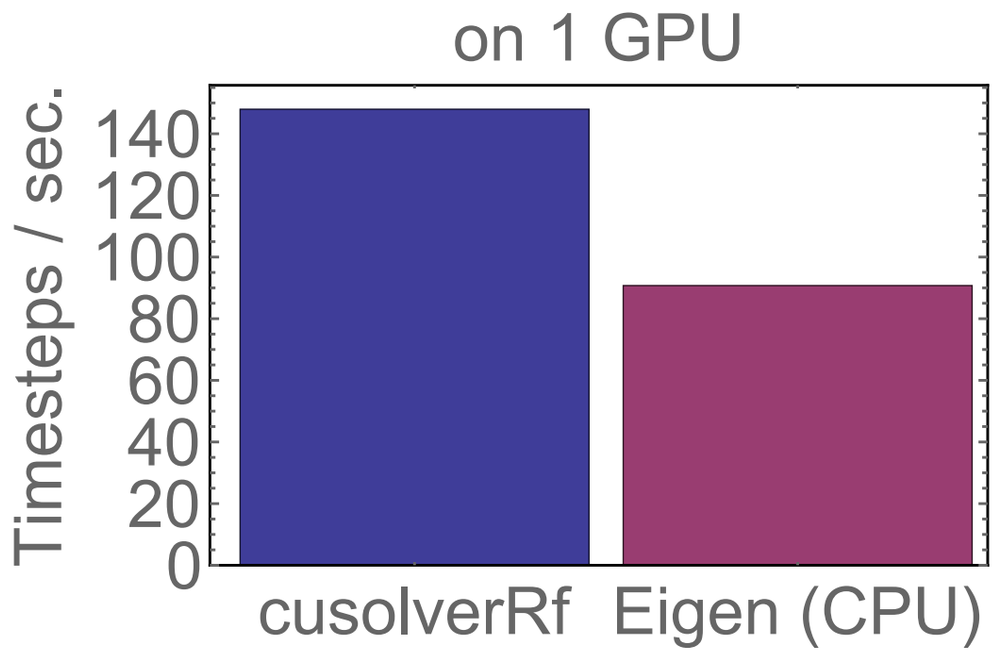
λ: vector of Lagrange multipliers

M is sparse and the location of non-zeros in **M** does not change

⇝Solve on GPU using sparse QR refactorization with cuSolverRfRefactor()
(available with CUDA Toolkit version ≧7.5)

MPI implementation: dynamically update ghost layer width to include largest constraint cluster

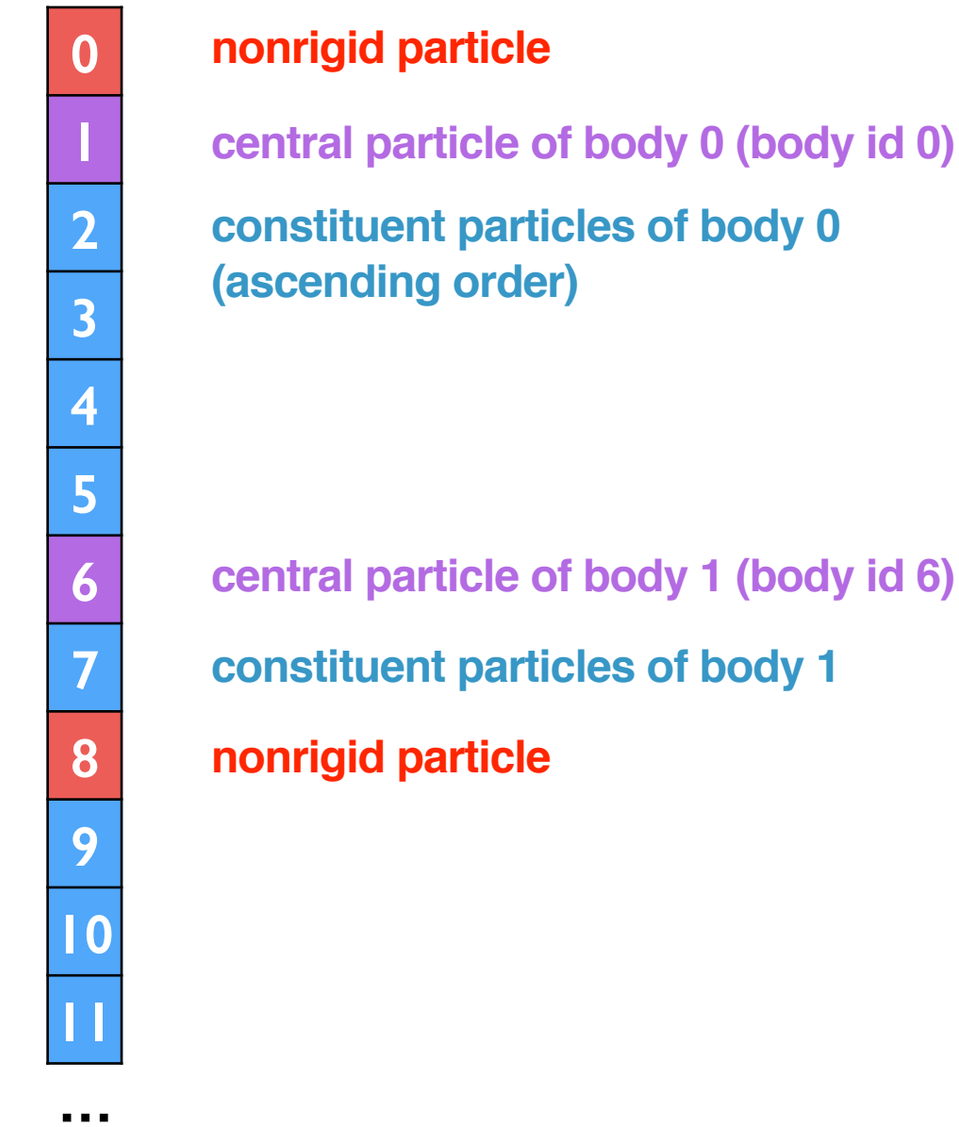
Protein aggregation benchmark

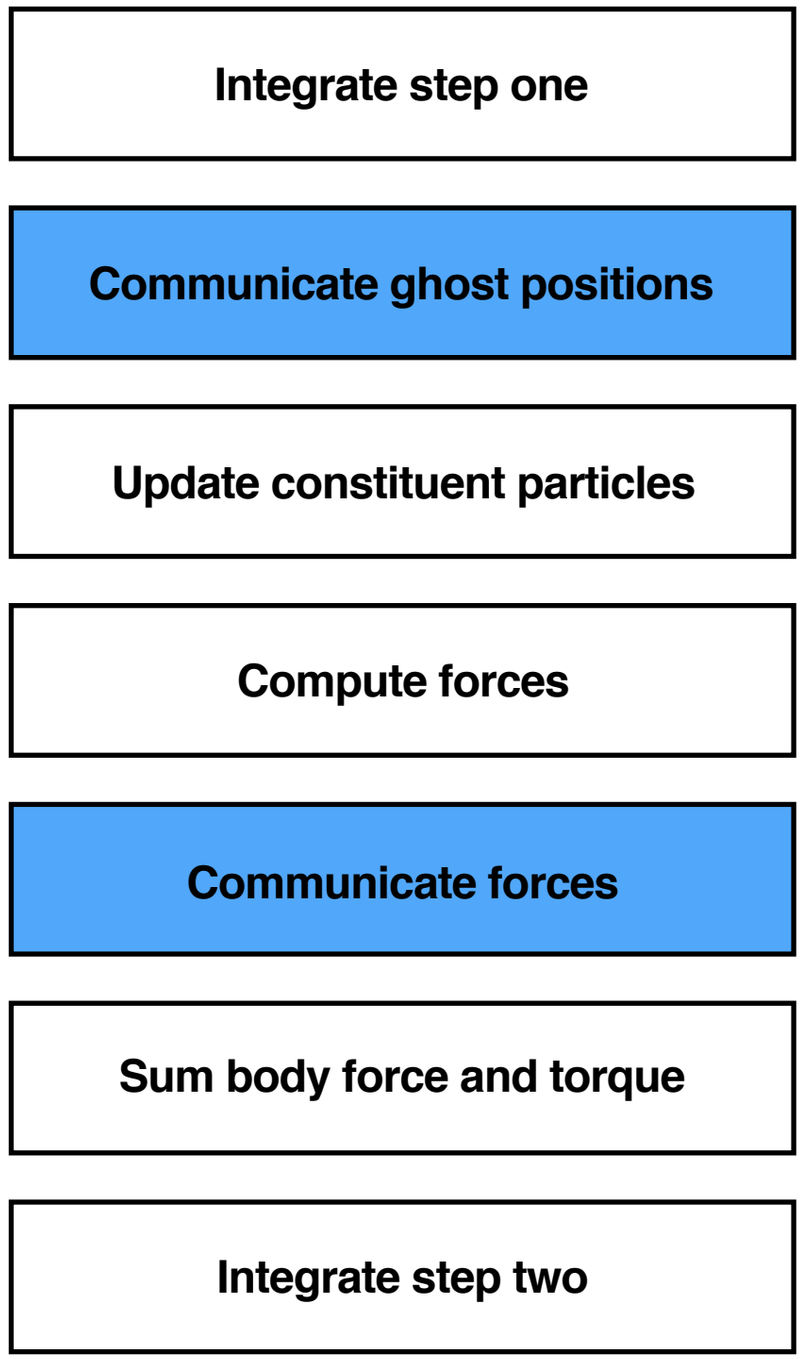


Composite bodies - md.constrain.rigid()

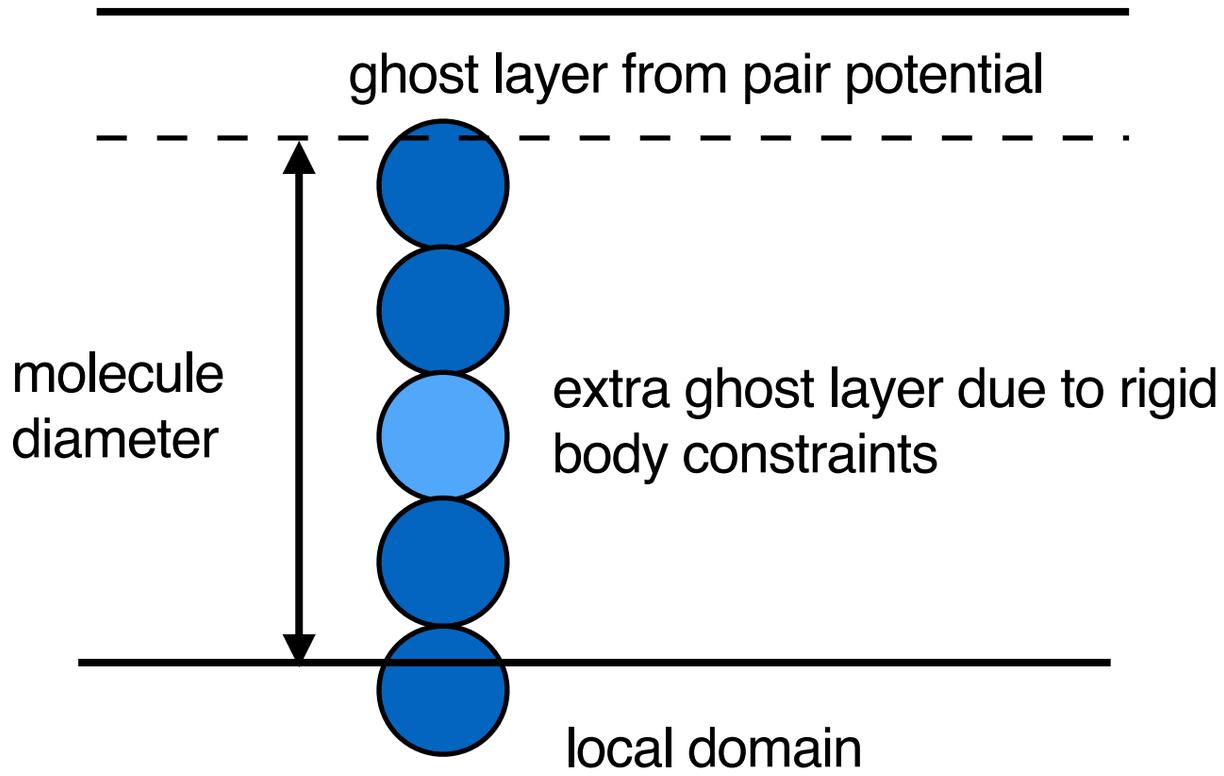
In HOOMD 2.0, composite body positions and orientations are tracked through their central particles

Central particles the same integrators (md.integrate.*) with non-rigid particles





Ghost layers for composite bodies

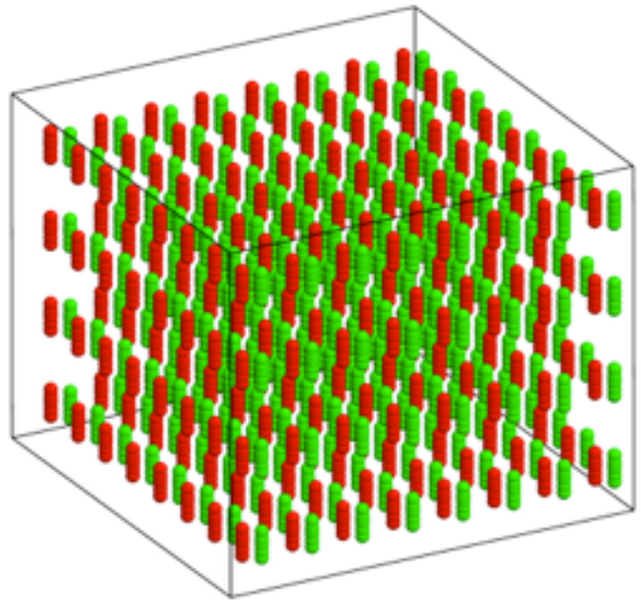


The central particle determines the positions of the the constituent particles

Ghost layer must be wide enough to ensure that *all* constituent particles are communicated whenever a part of the rigid body is within the interaction range

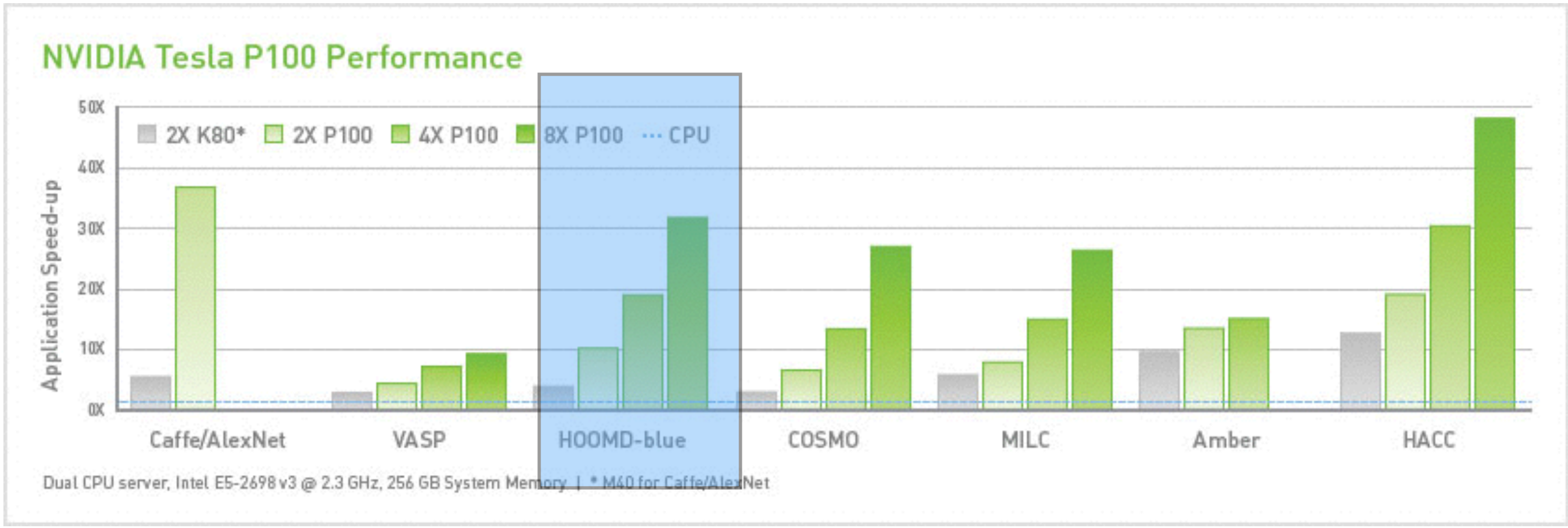
Example for constrain.rigid()

```
from hoomd import *
from hoomd import md

context.initialize()

# create rigid spherocylinders out of two particles (not including the central particle)
len_cyl = 2.5
n_bead = 5

uc = lattice.unitcell(N = 2, a1 = [4,0,0], a2 = [0,4,0], a3 = [0,0,len_cyl+4],position = [[0,0,0], [1,1,0]],
                      type_name = ['A', 'B'])
system = init.create_lattice(unitcell=uc, n=[8,8,4])

for p in system.particles:
    p.moment_inertia = (.5,.5,1)

# create constituent particle types
system.particles.types.add('A_const')
system.particles.types.add('B_const')

md.integrate.mode_standard(dt=0.001)

# central particles
lj = md.pair.lj(r_cut=False,nlist=md.nlist.cell())
lj.pair_coeff.set(['A','B'], system.particles.types, epsilon=1.0, sigma=1.0, r_cut=2.5)

# constituent particle coefficients
lj.pair_coeff.set('A_const','A_const', epsilon=1.0, sigma=1.0, r_cut=2**(1./6.))
lj.pair_coeff.set('B_const','B_const', epsilon=1.0, sigma=1.0, r_cut=2**(1./6.))
lj.pair_coeff.set('A_const','B_const', epsilon=1.0, sigma=1.0, r_cut=2.5)

rigid = md.constrain.rigid()
rigid.set_param('A', types=['A_const']*n_bead, positions=[(0,0,-len_cyl/2+i*len_cyl/n_bead) for i in range(n_bead)])
rigid.set_param('B', types=['B_const']*n_bead, positions=[(0,0,-len_cyl/2+i*len_cyl/n_bead) for i in range(n_bead)])

# create the constituent particles
rigid.create_bodies()
center = group.rigid_center()

langevin = md.integrate.langevin(group=center,kT=1.0,seed=123)
langevin.set_gamma('A',2.0)
langevin.set_gamma('B',2.0)
run(1e7)
```



Oppositely charged rigid rods

Performance on Pascal P100



HPMC Dodecahedron benchmark

http://www.nvidia.com/object/tesla-p100.html

Summary

- HOOMD-blue is flexible, python-based and optimized for latest GPU generations, all major features available with MPI
- In HOOMD-blue 2.0, PPPM electrostatics, distance constraints and rigid body constraints are supported in multi-GPU configuration
- Targeted for large-scale biomolecular self-assembly

Acknowledgments

ARO # W911NF-15-1-0185