UCG-MD: efficient “ultra-coarse-grained” molecular dynamics

John Grime

PI: Prof. Gregory A. Voth

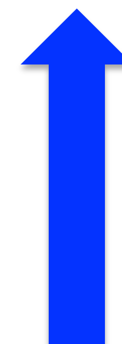
University of Chicago / Argonne National Laboratory

NCSA Blue Waters Symposium, May 2014

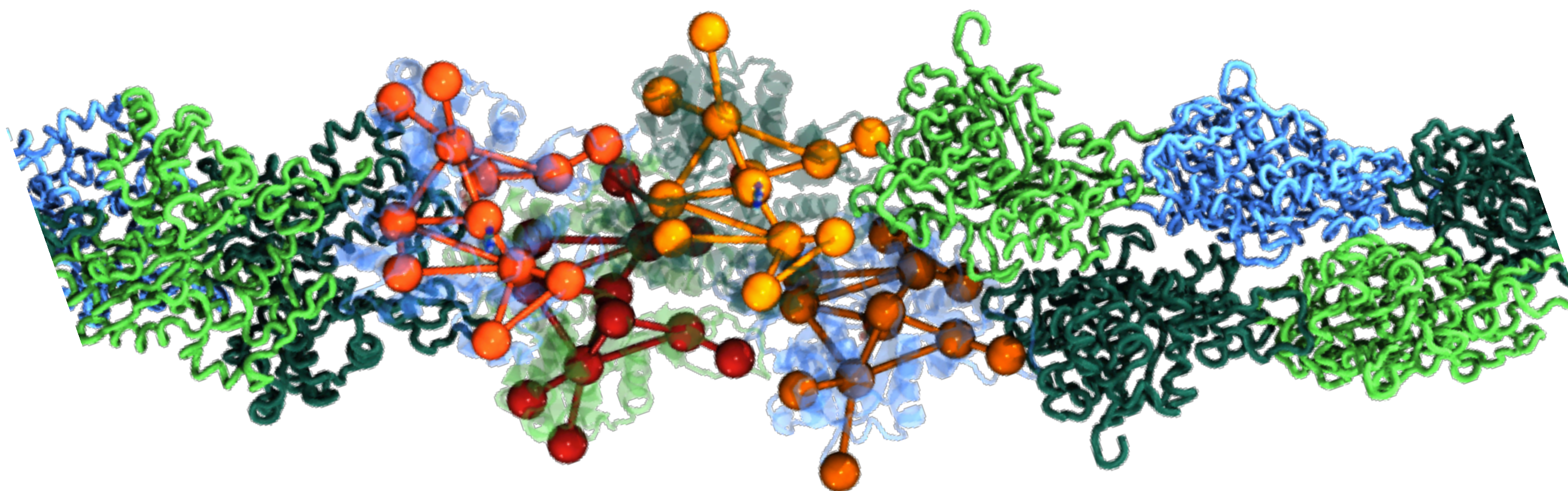
"Ultra coarse grained" (UCG) models

- "Ultra-coarse-grained" (UCG) model
- Highly coarse grained solvent free model
- Coarse grained solvent free model
- Atomistic solvent free model
- Atomistic model with solvent

UCG



Atomic model



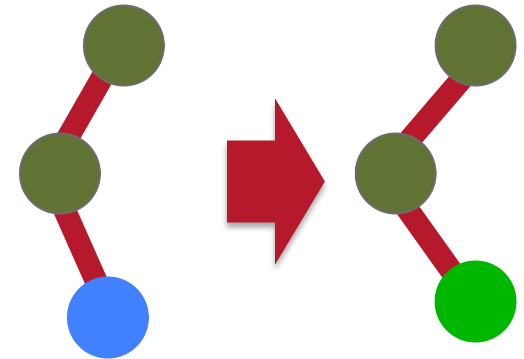
Dama et al, *J. Chem. Theory Comput.*, **2013**, 9, 2466-2480

“Ultra coarse grained” (UCG) models

“States” in the UCG “Beads”

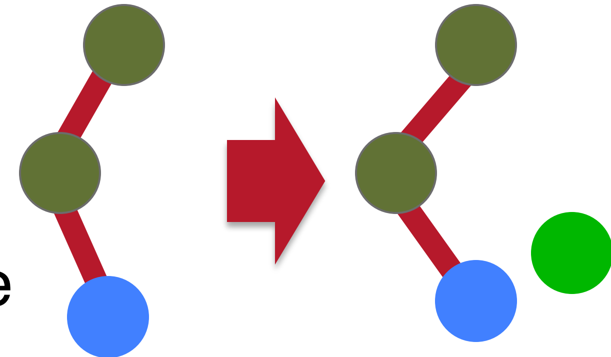
States within CG beads:

— physical —	**— chemical —**
loop folding	redox reaction
hydrophobic collapse	isomerization
ligand binding	protonation



State-dependent CG interactions:

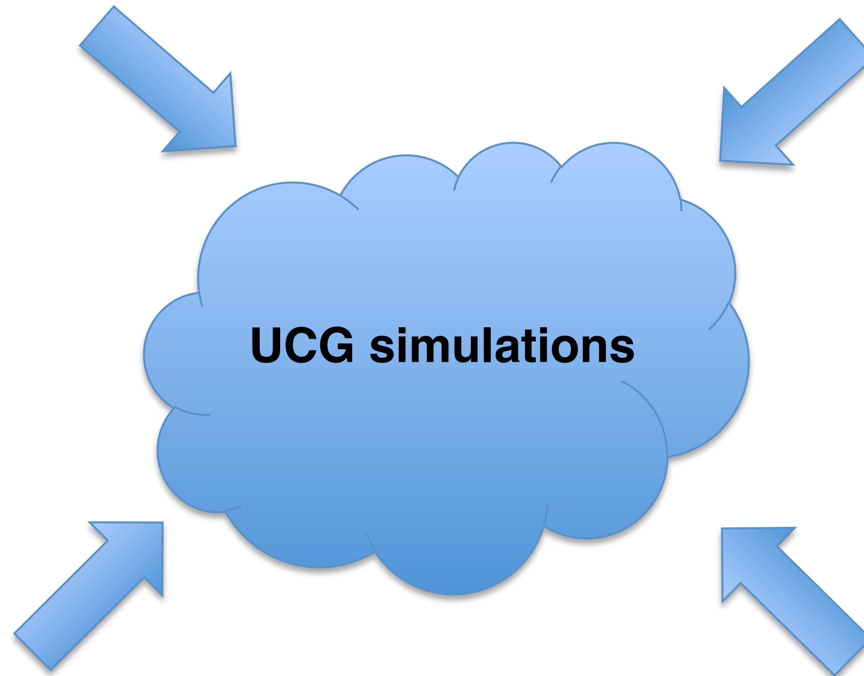
— physical —	**— chemical —**
protein folding	phosphorylation
domain folding	enyzmatic cleavage



UCG simulations: what do we need?

Dynamic system components
(significant runtime changes)

Low memory requirements
(very large simulations)



Dynamic load balancing
(implicit solvent etc)

Flexibility
(ease of modification)

Existing codes: typically one or more of the above, but not all.

UCG-MD

- Basic design principles:
  - All communications **nonblocking** where possible (incl. collectives!)
  - **Parallel IO** where possible (problems with MPIIO?)
  - Internal data = **flat arrays** where possible (GPU, OpenCL etc)
  - Portable, self-contained (C++ & MPI, *some* DMAPP/PAMI)
  - **Self profiling** (automatic parallel timing/imbalance summary!)

- This talk will cover some specific details:
  - Use of **“template” data** (enables dynamic simulation contents)
  - Key **memory reduction** techniques (useful for very large systems)
  - **Load balancing** (parallel efficiency)

J. M. A. Grime and G. A. Voth,
“Highly Scalable and Memory-Efficient Ultra-Coarse-Grained Molecular Dynamics Simulations”,
J. Chem. Theory Comput., **2014**, 10, 423-431

Dynamic system components

- Template **subunits:**
  - Simplest form: subunit = molecule!
  - Member particle types, local topology (bonds, angles, etc)

- Template **assemblies:**
  - Member subunit types, additional local topology

- Topology information etc **generated dynamically**, at runtime:
  - No global bond/angle/dihedral list etc
  - No global "particle indices"
  - No global nonbonded exclusion lists (1-2, 1-3, etc)

As most information is generated dynamically, input files are small, simple: simulation data can be modified extensively at runtime (add/remove molecules, change molecular topologies and particle properties etc)

Dynamic system components

```
atom     0   name=a0  mass=10
atom     1   name=a1  mass=10
atom     2   name=a2  mass=10

register topo type=harmonic_bond  name=hb
register topo type=harmonic_angle name=ha

subunit 0 MySubunit
      member   type=a0   name=one
      member   type=a1   name=two
      member   type=a2   name=three

      topo hb  one two   parameters K=1 r0=4.6
      topo hb  two three parameters K=2 r0=3.6

      topo ha  one two three parameters K=10 theta0=180
end

assembly 0 MyAssembly
      member type=MySubunit name=s1
      member type=MySubunit name=s2

      topo hb   s1.three s2.one  parameters K=1 r0=2

      topo ha   s1.two s1.three s2.one parameters K=10 theta0=180
end
```

Anatomy of a simple template definition file …

Dynamic system components

```
atom      0    name=a0  mass=10
atom      1    name=a1  mass=10
atom      2    name=a2  mass=10

register topo type=harmonic_bond  name=hb
register topo type=harmonic_angle name=ha

subunit 0 MySubunit
      member   type=a0   name=one
      member   type=a1   name=two
      member   type=a2   name=three

      topo hb  one two   parameters K=1 r0=4.6
      topo hb  two three parameters K=2 r0=3.6

      topo ha  one two three parameters K=10 theta0=180
end

assembly 0 MyAssembly
      member type=MySubunit name=s1
      member type=MySubunit name=s2

      topo hb   s1.three s2.one  parameters K=1 r0=2

      topo ha   s1.two s1.three s2.one parameters K=10 theta0=180
end
```

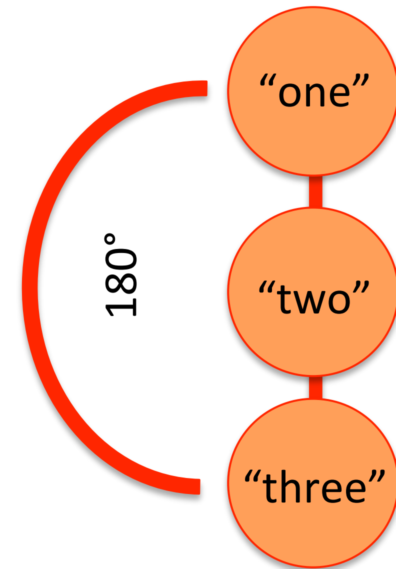
Define particle types we'll use

Define topology types we'll use (with shorthand names!)

Dynamic system components

```
atom      0    name=a0   mass=10
atom      1    name=a1   mass=10
atom      2    name=a2   mass=10

register topo type=harmonic_bond  name=hb
register topo type=harmonic_angle name=ha

subunit 0 MySubunit
      member   type=a0   name=one
      member   type=a1   name=two
      member   type=a2   name=three

      topo hb  one two   parameters K=1 r0=4.6
      topo hb  two three parameters K=2 r0=3.6

      topo ha  one two three parameters K=10 theta0=180
end

assembly 0 MyAssembly
      member type=MySubunit name=s1
      member type=MySubunit name=s2

      topo hb   s1.three s2.one  parameters K=1 r0=2

      topo ha   s1.two s1.three s2.one parameters K=10 theta0=180
end
```

Define a subunit, "MySubunit":

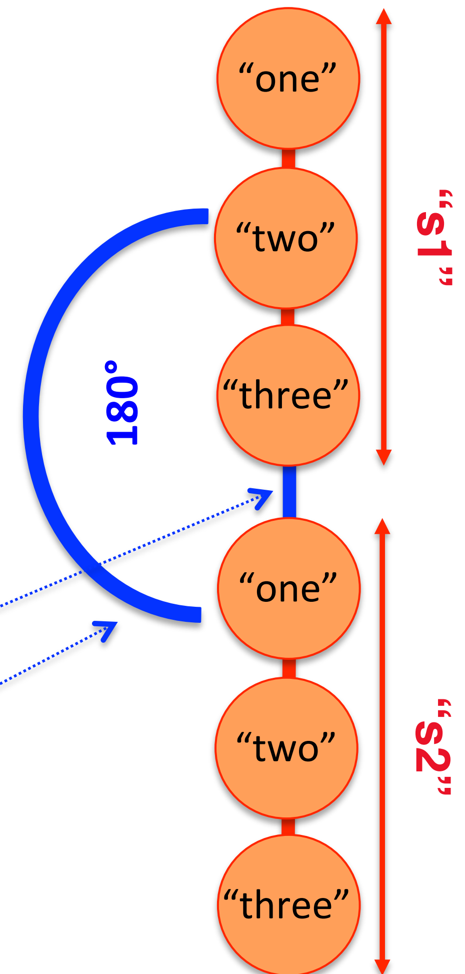


Dynamic system components

Define an assembly, "MyAssembly":

```
atom      0    name=a0   mass=10
atom      1    name=a1   mass=10
atom      2    name=a2   mass=10

register topo type=harmonic_bond  name=hb
register topo type=harmonic_angle name=ha

subunit 0 MySubunit
      member   type=a0    name=one
      member   type=a1    name=two
      member   type=a2    name=three

      topo hb   one two    parameters K=1 r0=4.6
      topo hb   two three parameters K=2 r0=3.6

      topo ha   one two three parameters K=10 theta0=180
end
```

```
assembly 0 MyAssembly
      member type=MySubunit name=s1
      member type=MySubunit name=s2

      topo hb   s1.three s2.one  parameters K=1 r0=2

      topo ha   s1.two s1.three s2.one parameters K=10 theta0=180
end
```

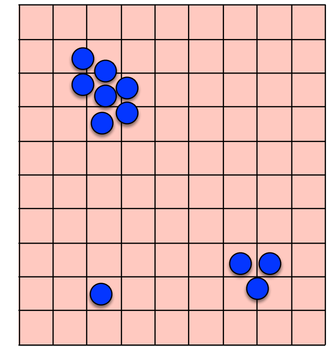


... etc

Memory use: sparse data structures

How do we know which particles are interacting?

Test all particles: $O(N^2)$, **bad idea as N gets large!**



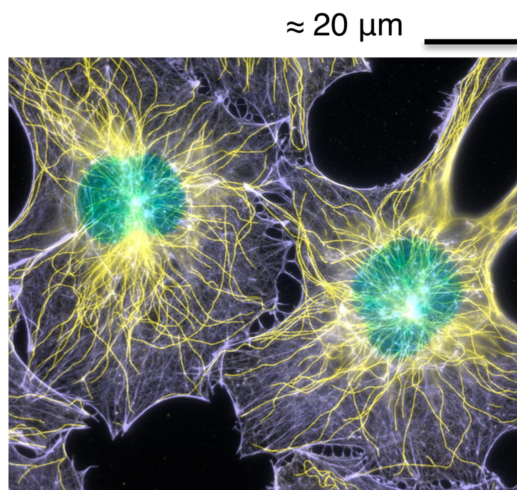
Conventional "link-cells"

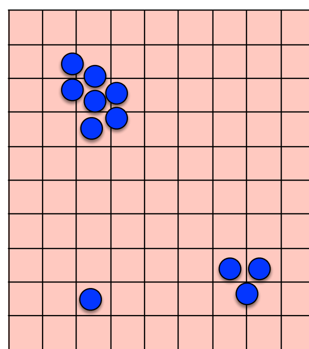
Verlet lists – store lists of particles close enough to interact. Positions are temporally correlated, so lists reused for several timesteps. Still $O(N^2)$ to generate, but cost amortized over several steps.

Improvement: **Link cells** used to generate Verlet lists. Map particles into a lattice, iterate over cell neighbors in 3D. More efficient! **But …**

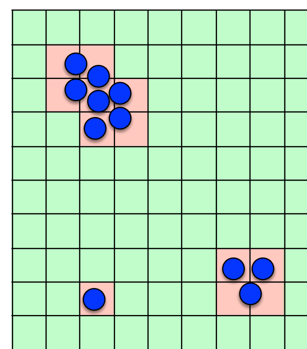
Memory use: sparse data structures

Biological/materials systems could be multiscale over *many orders of magnitude: e.g.* **Å → µm**

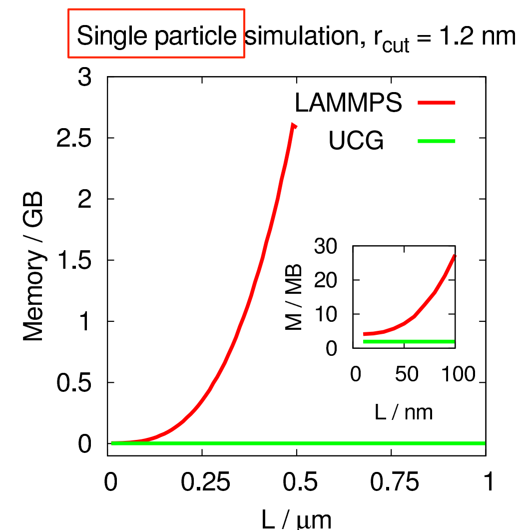




Conventional algorithms



Sparse algorithms



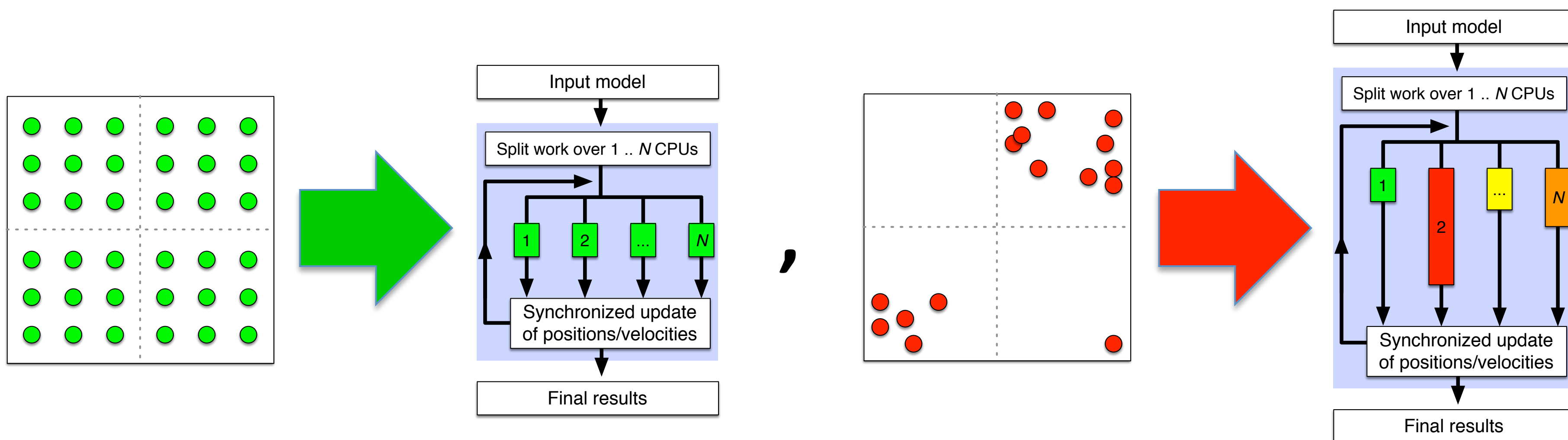
Even a **single particle** with conventional algorithms: huge memory to span these multiple length scales. *Only use memory where needed!*
Rather than a “complete” flat array of link cells, we use a dynamic tree indexed with key = (x,y,z), link cell lattice coordinate

Mouse fibroblast cells, Torsten Wittman, The Scripps Research Institute

Load balancing

Parallel MD “tightly coupled”: overall simulation rate is limited by the CPU with the most work to do. Need to balance the workload …

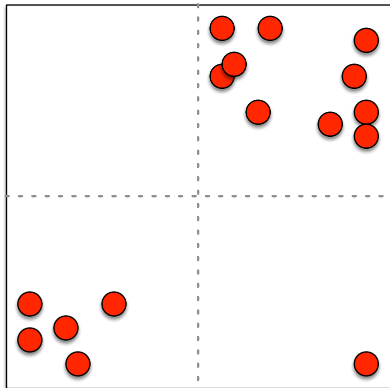
With **~uniform** particle density (e.g. *explicit* solvent): split simulation into equal volumes per CPU - **load balancing emerges naturally**



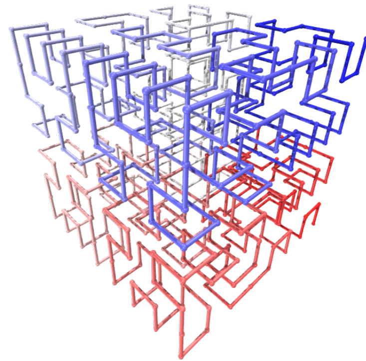
With **non-uniform** particle density (e.g. *implicit* solvent): naïve use of the same approach does not work as well - **load imbalance**

Load balancing

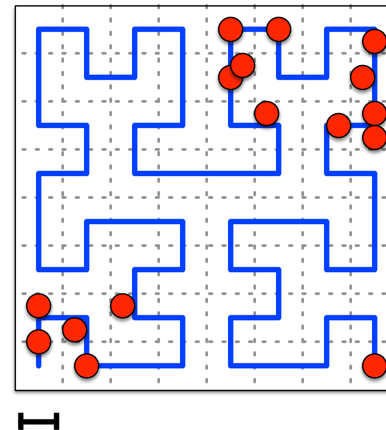
Load balancing via a *Hilbert space-filling curve (SFC):*



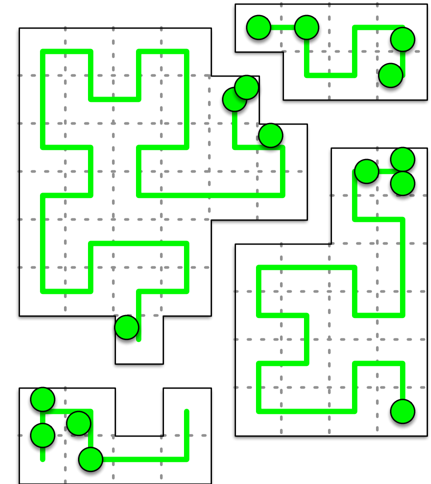
Unbalanced system



Apply SFC



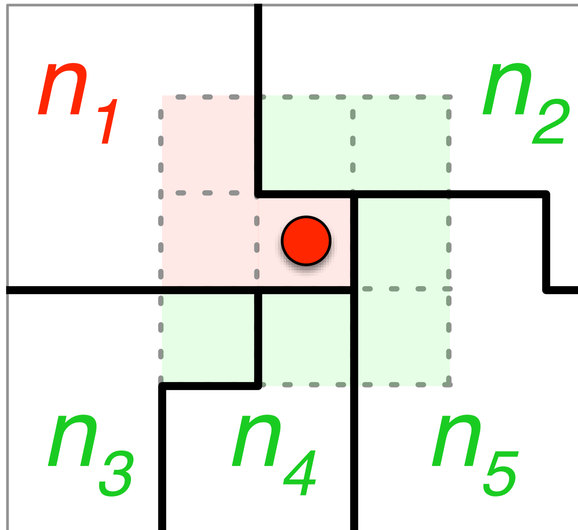
Map particles into SFC



Section SFC

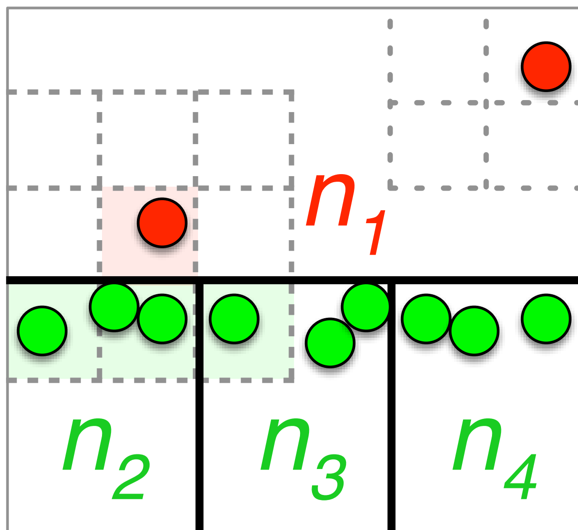
Approach borrowed from astrophysics: Hilbert SFC allows reversible mapping of 3D lattice coordinates into a 1D “curve index”. Curve is then sectioned for roughly equal load in each section, sections then assigned to CPUs. (Locality of data, compression, …)

Load balancing



Who to talk to: Hilbert SFC sections can be very irregular volumes, sharing interfaces with variable numbers of adjacent domains. *Dynamic* at runtime. Each CPU therefore uses **DMAPP "remote memory access"** to inform other CPUs to expect incoming shared particle data (very efficient!)

eg: n_1 informs $n_{2\text{-}5}$ to expect communication

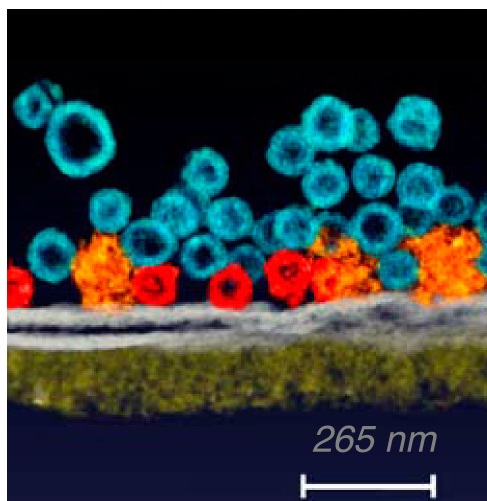


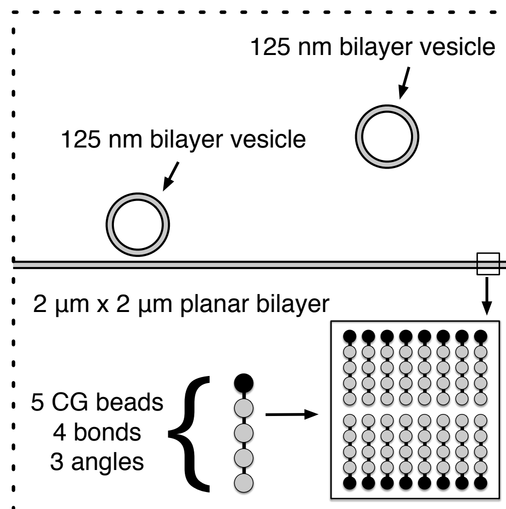
What to say: Hilbert SFC could potentially assign very large spatial volumes to CPUs - with very large surface areas. **Prefilter particle data** before sharing across interfaces: communicate only that particle data which is actually needed by the remote CPU.

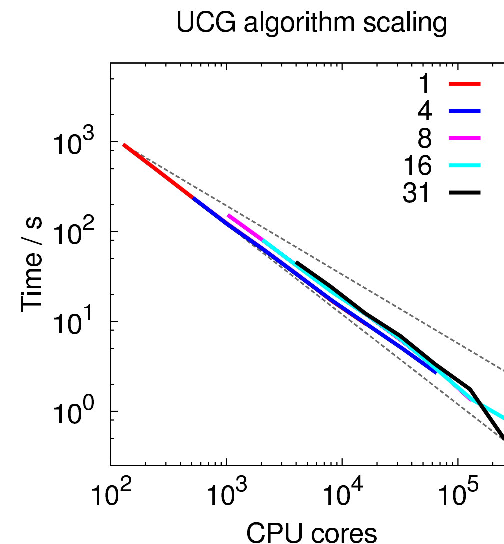
eg: only *some* particles shared between $n_{2\text{-}4}$ and n_1

Load balancing

Large planar CG bilayer (2 μm x 2 μm)
Two spherical CG bilayers (*d* = 125 nm)







Despite *extremely* heterogeneous particle distribution and lightweight computation (**~5x fewer** pair interactions per particle vs all-atom), algorithms seem to scale to **~260,000 CPU cores** on Blue Waters (at which point the DMAPP libraries failed intermittently)

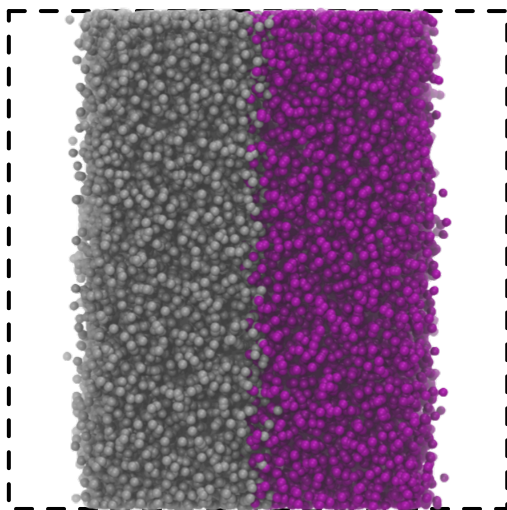
Synaptic tomogram: Zampighi et al, *Biophys. J.* **2006**, 91, 2910-2918

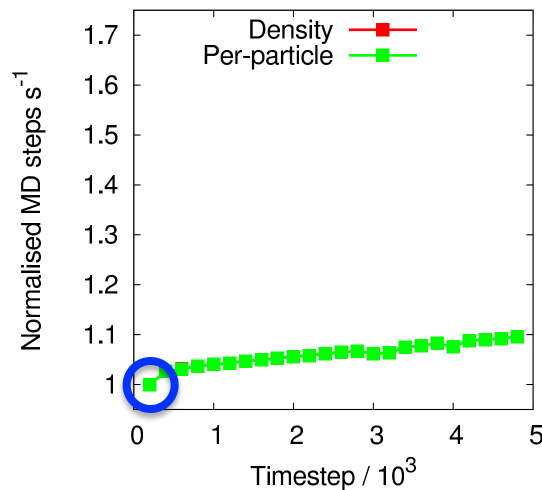
Load balancing

- CG/UCG: **heterogeneous** interactions in the same simulation?
  - Lennard-Jones, Gay-Berne, Yukawa, tabulated, …
  - Bonds, angles, dihedrals, …
  - Potentially exotic: *n*-body nonbonded, environmental dependence, …
- Different CG/UCG particles could have very different computational costs, so:
  - time *everything*
  - accumulate per-particle “cost” (travels with particles)
  - Feed costs into dynamic load balancer instead of assuming uniform cost for each particle

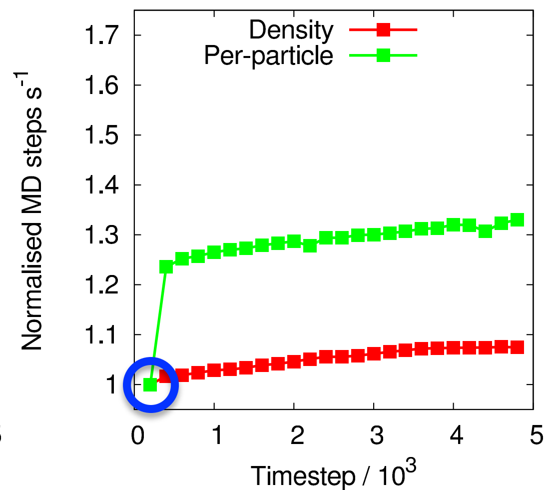
Load balancing: particle density vs *per-particle* timings

Simple 2-component system:



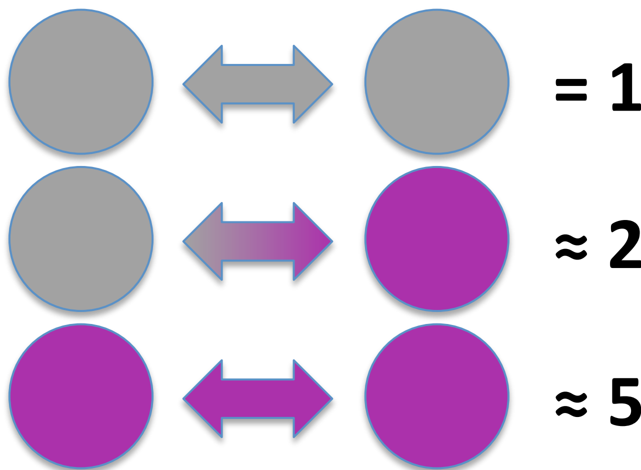


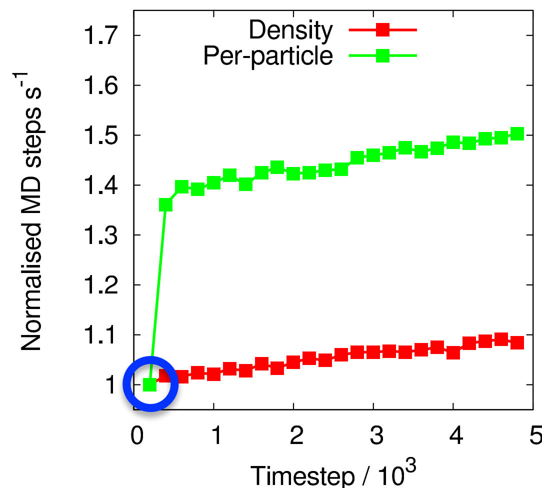
Single core



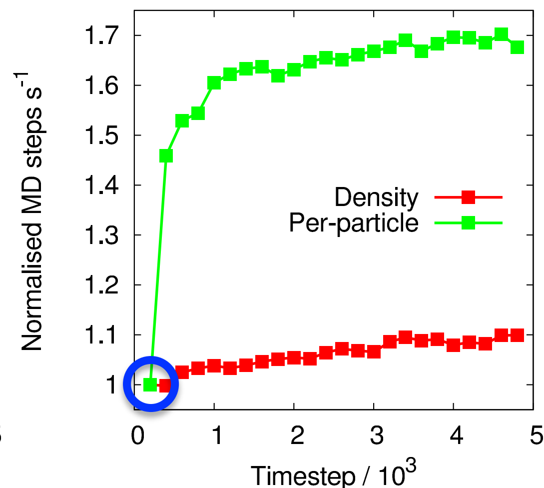
2 cores

Relative interaction costs:





4 cores



8 cores

Summary

- Currently using UCG-MD on Blue Waters for:
  - CG Protein self-assembly
  - CG membrane dynamics / remodeling
- Future enhancements:
  - GPU/CPU agnostic acceleration (OpenCL)
  - Improve communications efficiency (more DMAPP/PAMI)
- Acknowledgements:

  - NSF / NCSA
  - Voth group (special mention to James Farris Dama!)
  - **Blue Waters point-of-contact: Robert Brunner**