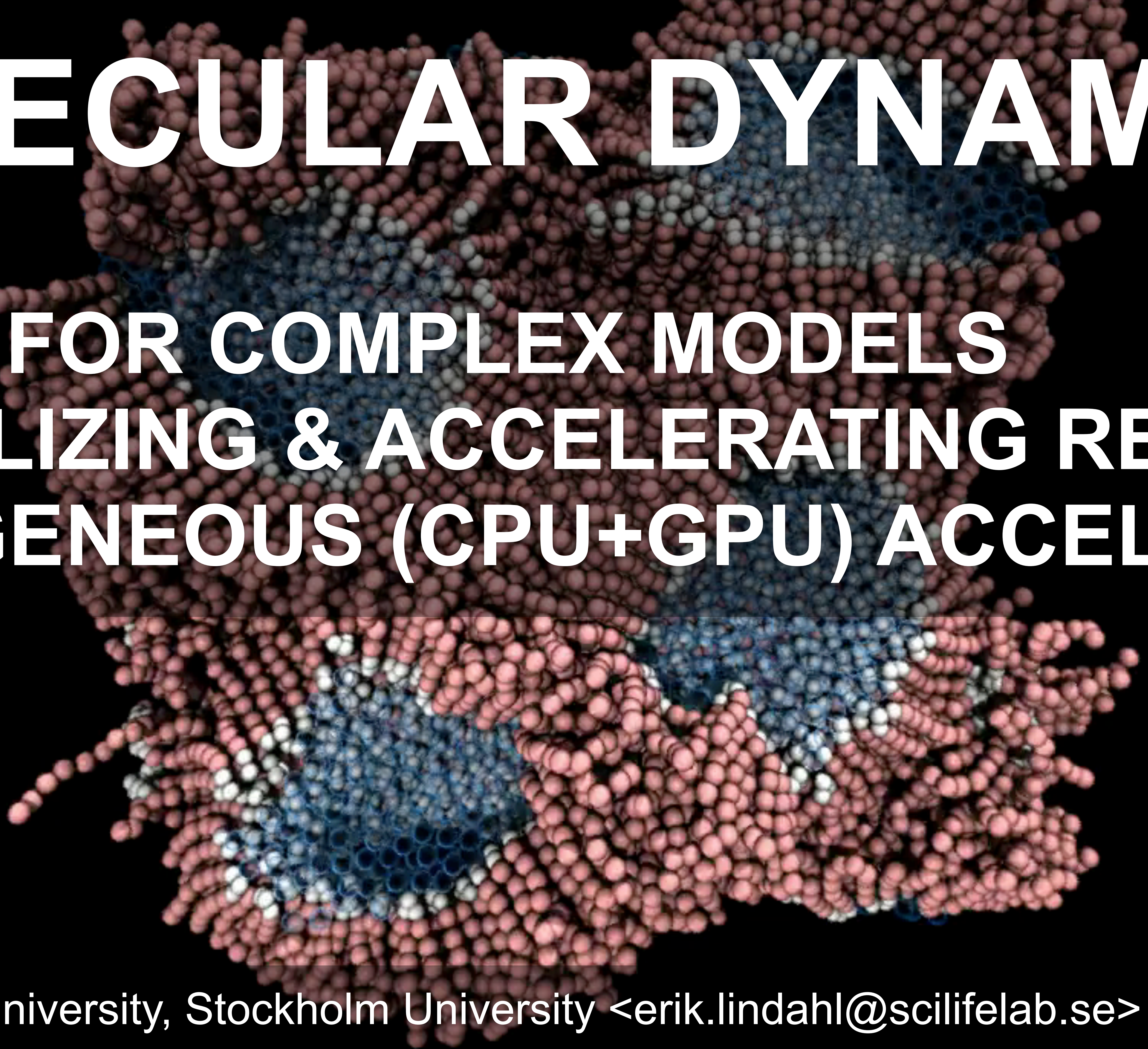
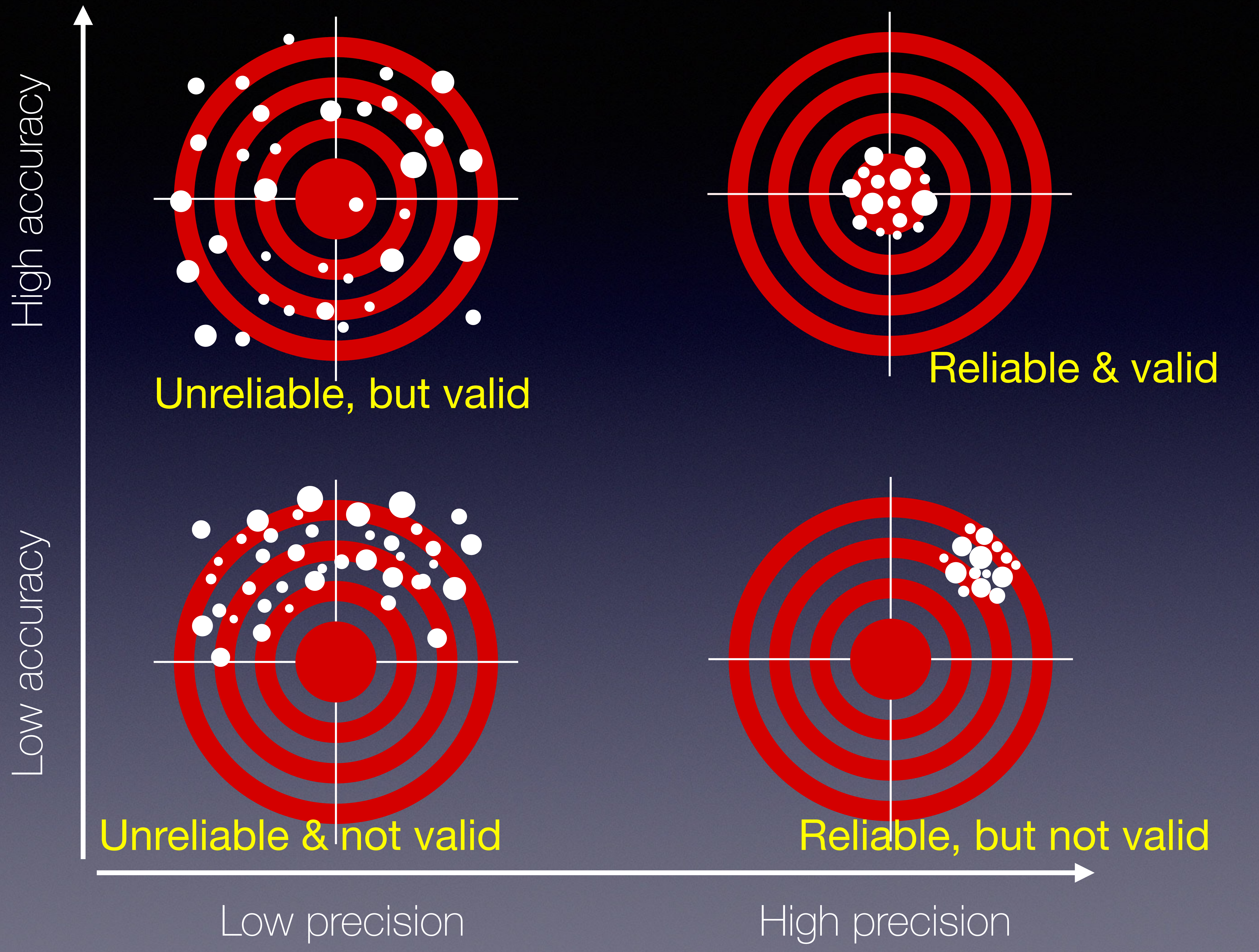
HPC IN LIFE SCIENCES: MOLECULAR DYNAMICS



1. CAVEATS FOR COMPLEX MODELS
2. PARALLELIZING & ACCELERATING REAL CODES
3. HETEROGENEOUS (CPU+GPU) ACCELERATION

Erik Lindahl, Stockholm University, Stockholm University <erik.lindahl@scilifelab.se>



High accuracy

Low accuracy

Unreliable, but valid

Reliable & valid

Unreliable & not valid

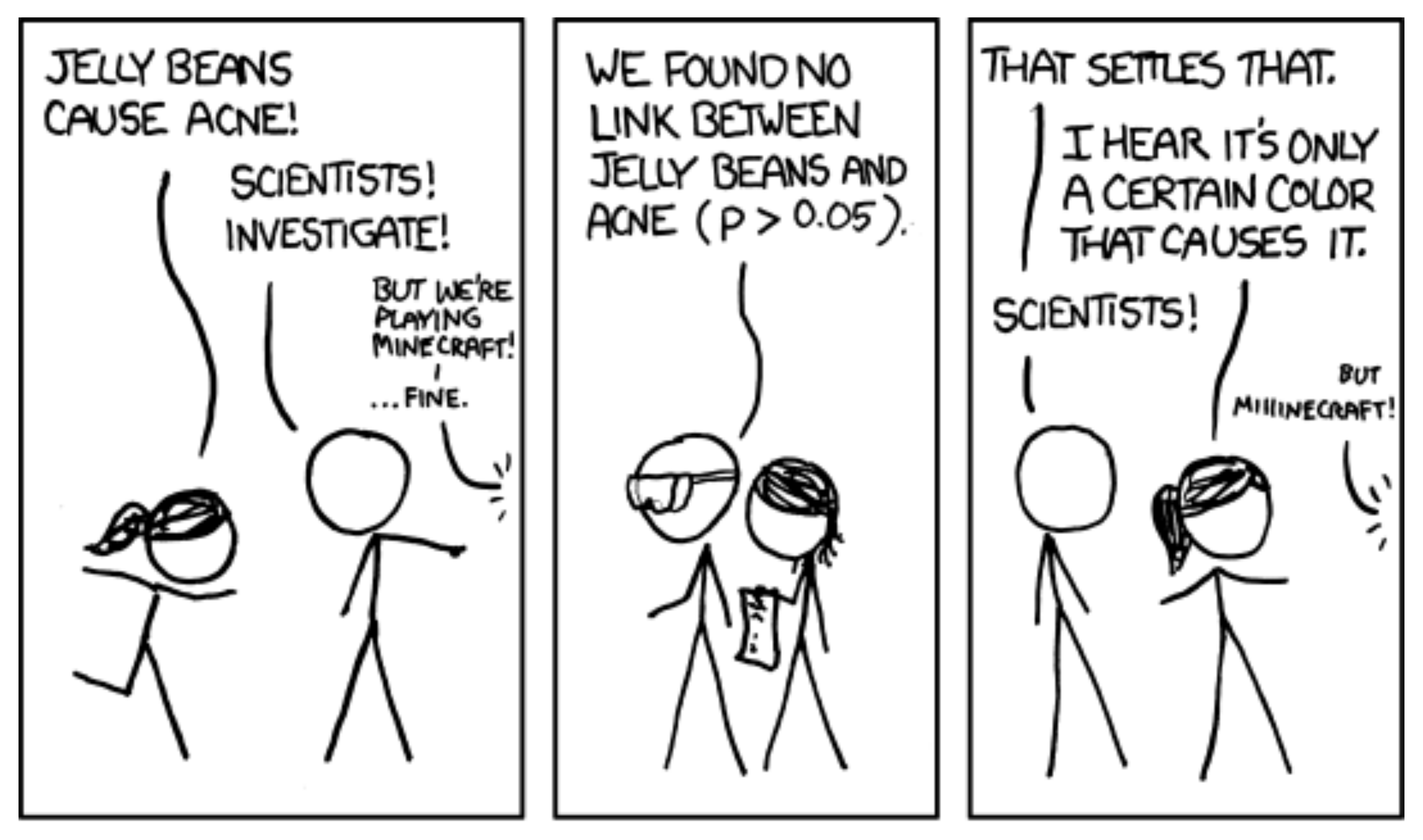
Reliable, but not valid

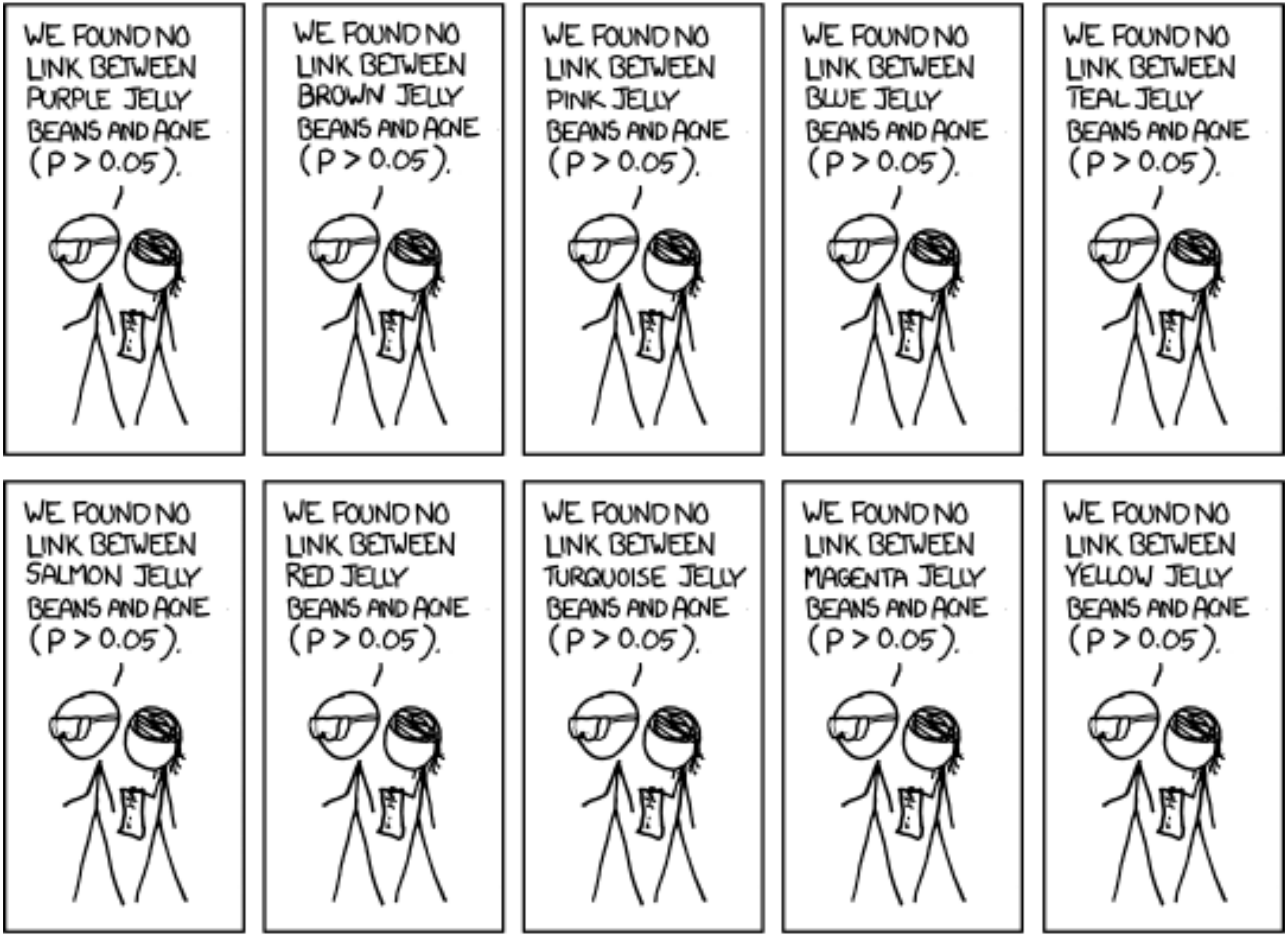
Low precision

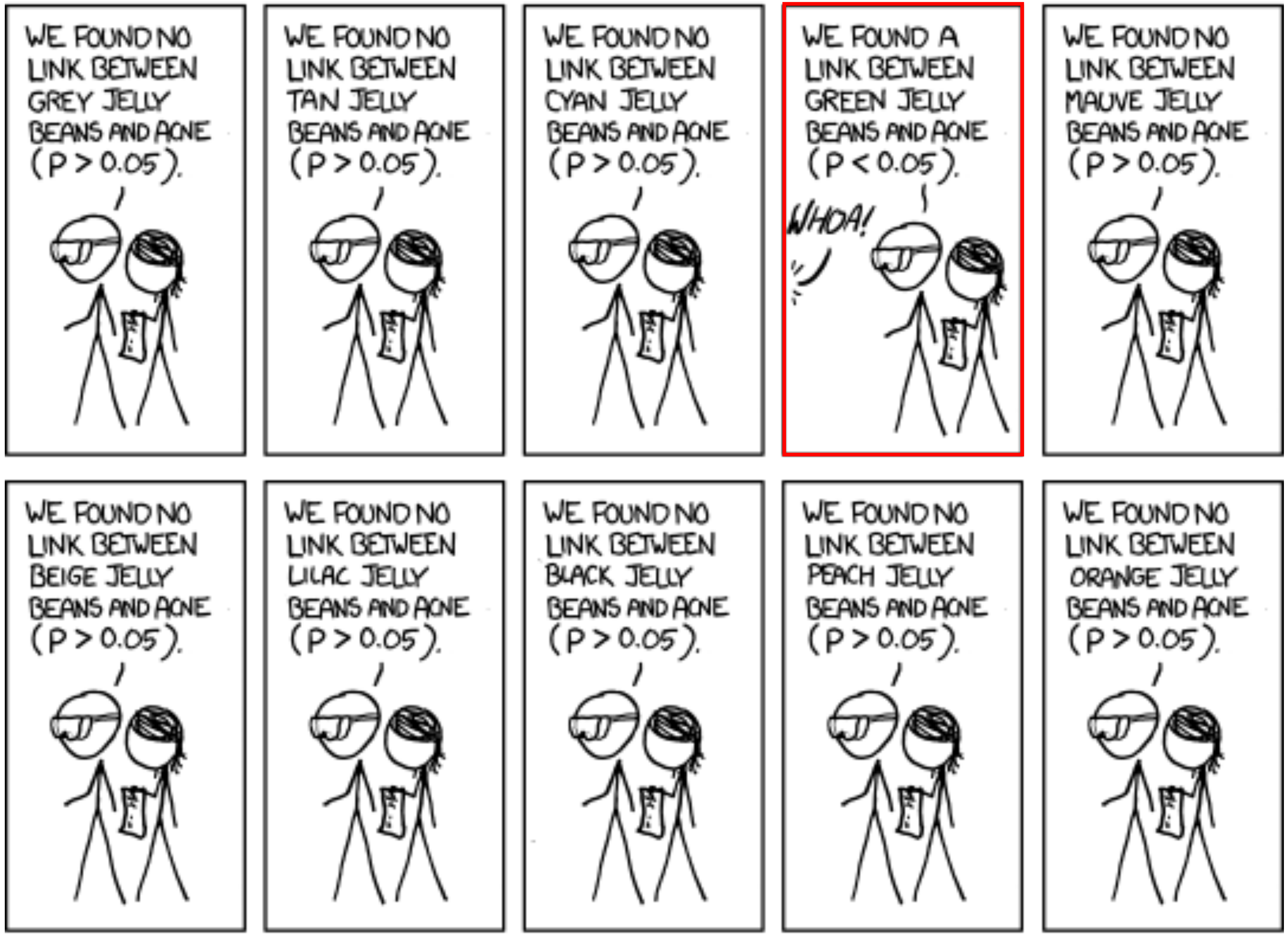
High precision

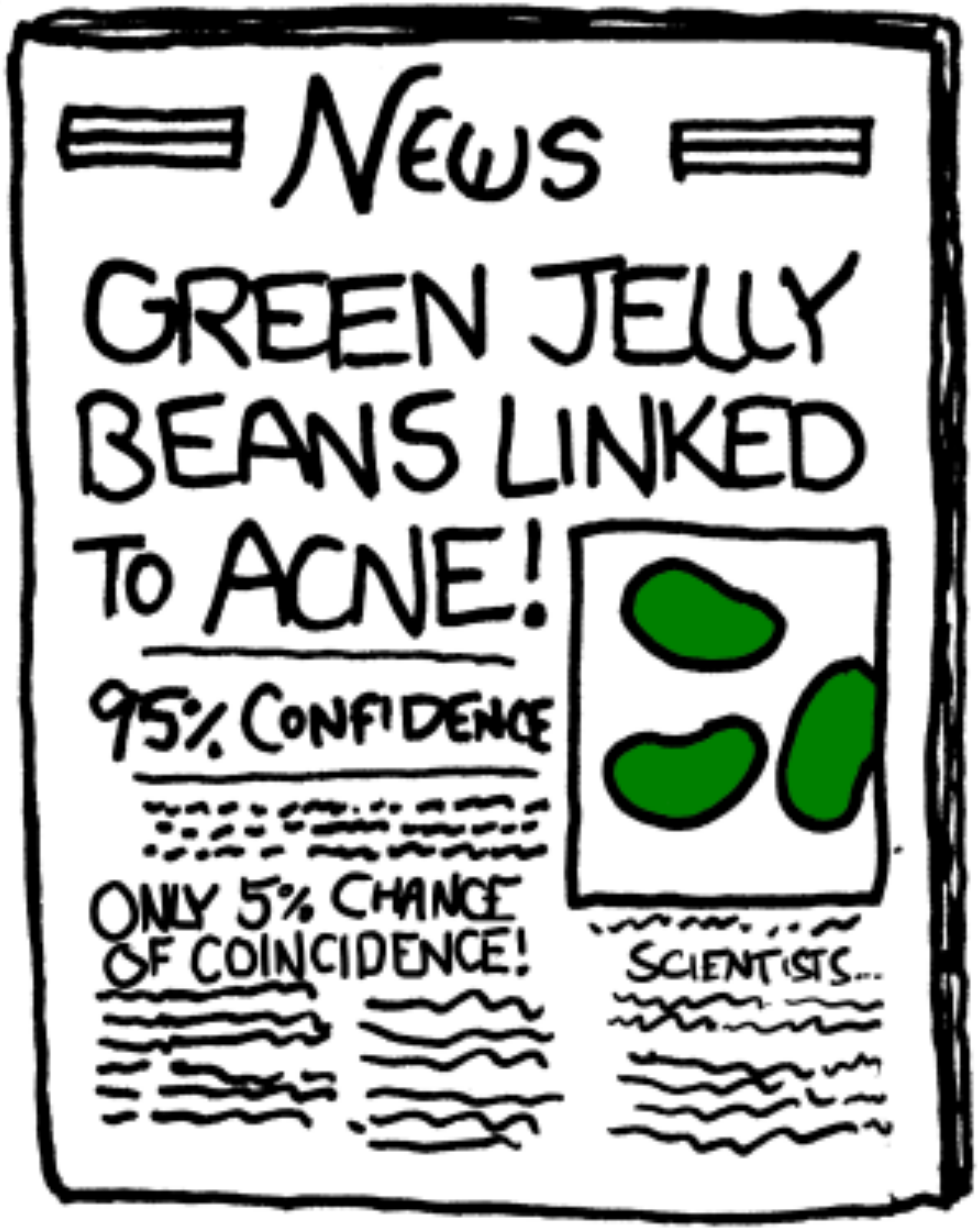
P-value hacking

https://www.xkcd.com/882/

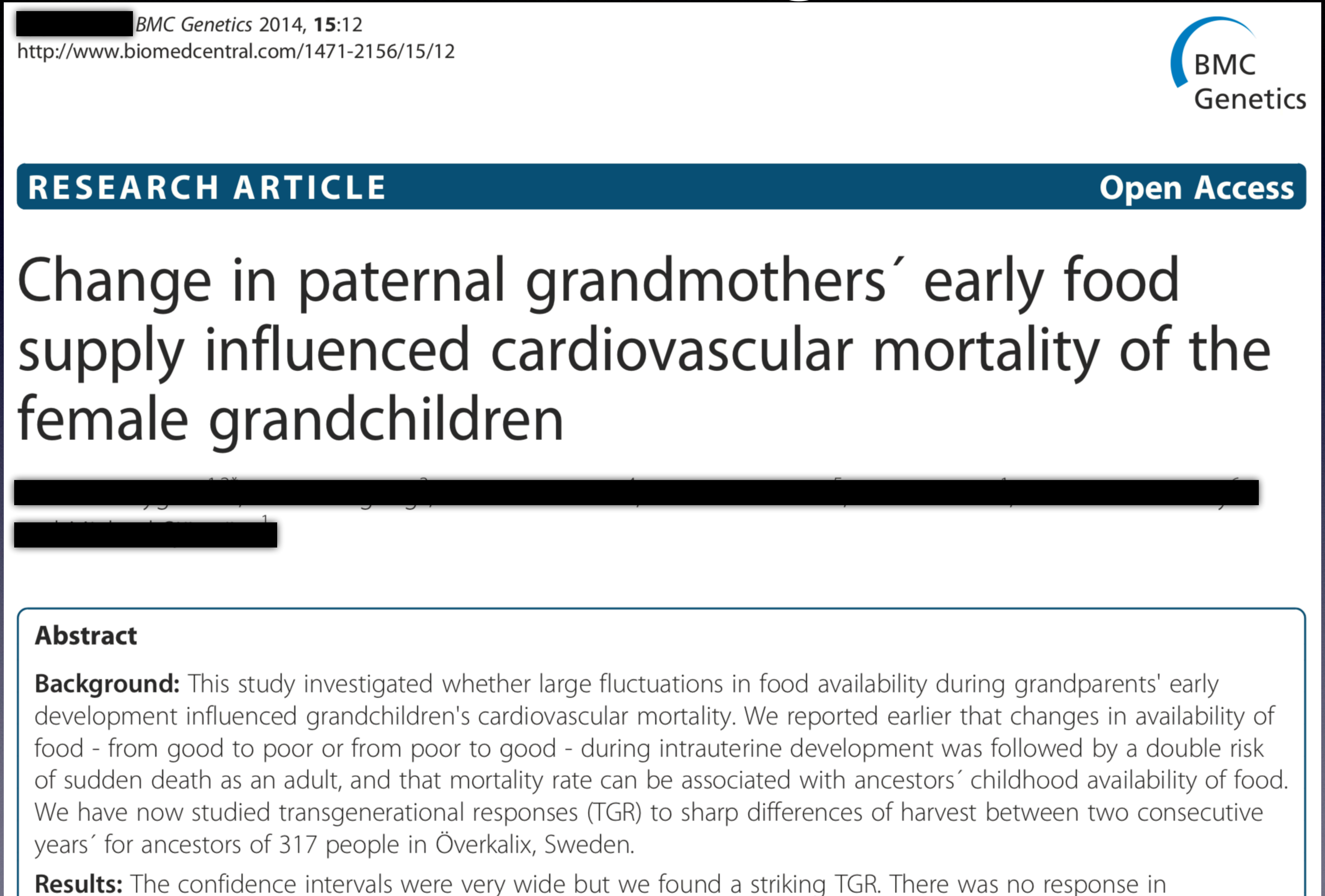

JELLY BEANS CAUSE ACNE!
SCIENTISTS! INVESTIGATE!
BUT WE'RE PLAYING MINECRAFT!
...FINE.
WE FOUND NO LINK BETWEEN JELLY BEANS AND ACNE (P > 0.05).
THAT SETTLES THAT.
I HEAR IT'S ONLY A CERTAIN COLOR THAT CAUSES IT.
SCIENTISTS!
BUT MIIIINECRAFT!


WE FOUND NO LINK BETWEEN PURPLE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN BROWN JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN PINK JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN BLUE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN TEAL JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN SALMON JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN RED JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN TURQUOISE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN MAGENTA JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN YELLOW JELLY BEANS AND ACNE (P > 0.05).


WE FOUND NO LINK BETWEEN GREY JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN TAN JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN CYAN JELLY BEANS AND ACNE (P > 0.05).
WE FOUND A LINK BETWEEN GREEN JELLY BEANS AND ACNE (P < 0.05).
WHOA!
WE FOUND NO LINK BETWEEN MAUVE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN BEIGE JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN LILAC JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN BLACK JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN PEACH JELLY BEANS AND ACNE (P > 0.05).
WE FOUND NO LINK BETWEEN ORANGE JELLY BEANS AND ACNE (P > 0.05).


News
GREEN JELLY BEANS LINKED TO ACNE!
95% CONFIDENCE
ONLY 5% CHANCE OF COINCIDENCE!
SCIENTISTS...

P-value hacking - for real



[redacted] *BMC Genetics* 2014, **15**:12
http://www.biomedcentral.com/1471-2156/15/12

RESEARCH ARTICLE **Open Access**

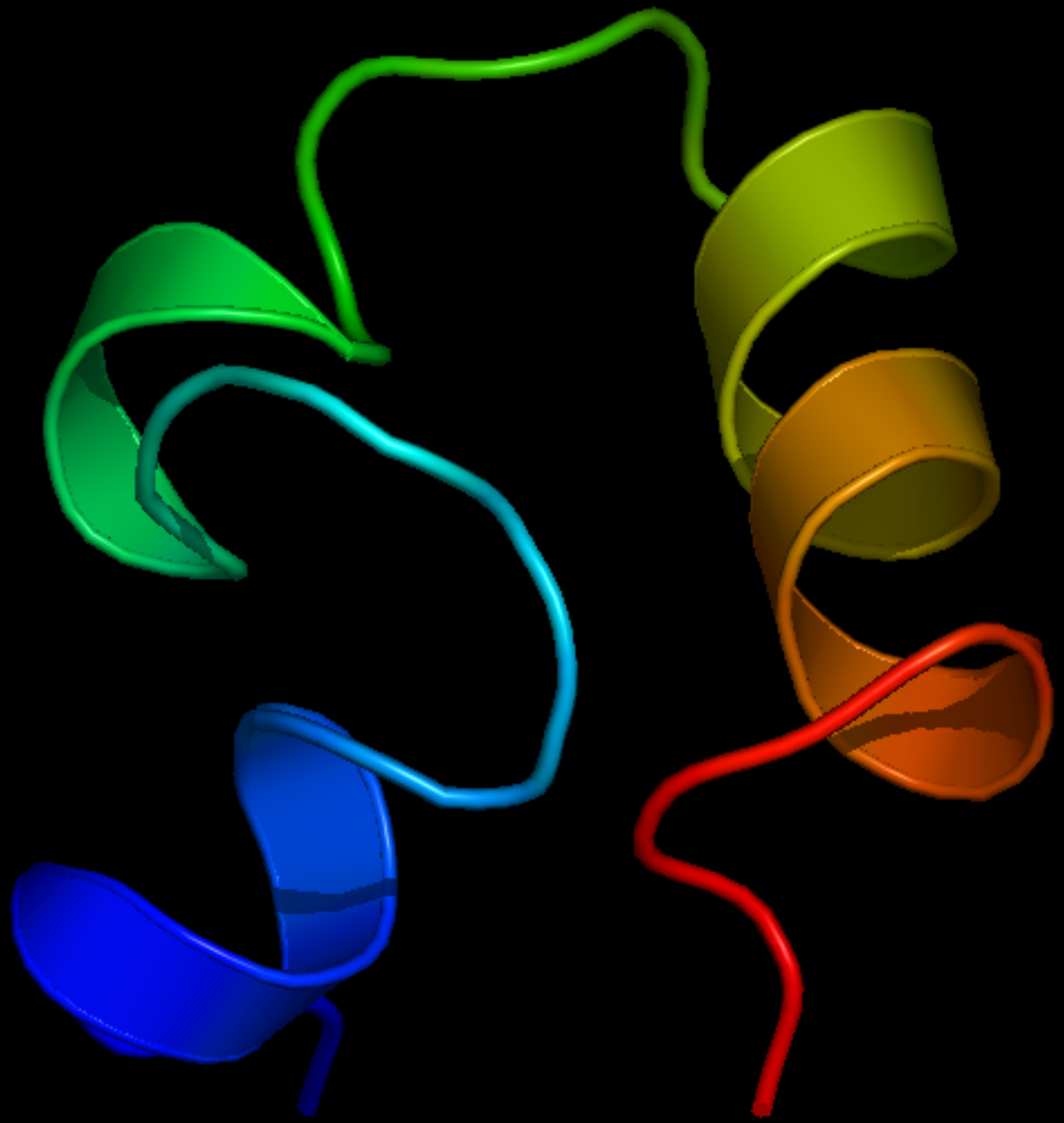
Change in paternal grandmothers´ early food supply influenced cardiovascular mortality of the female grandchildren

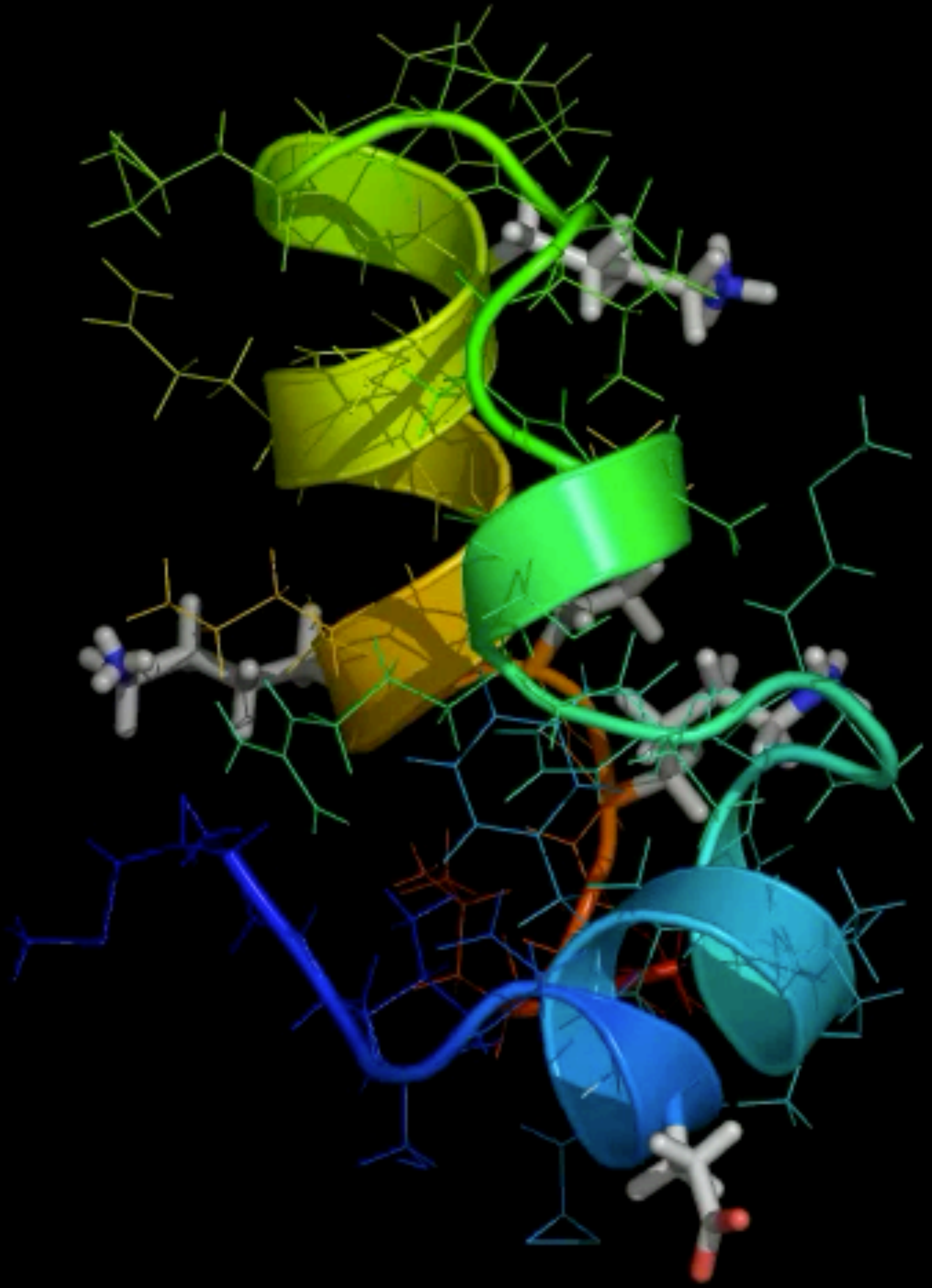
[redacted]

Abstract

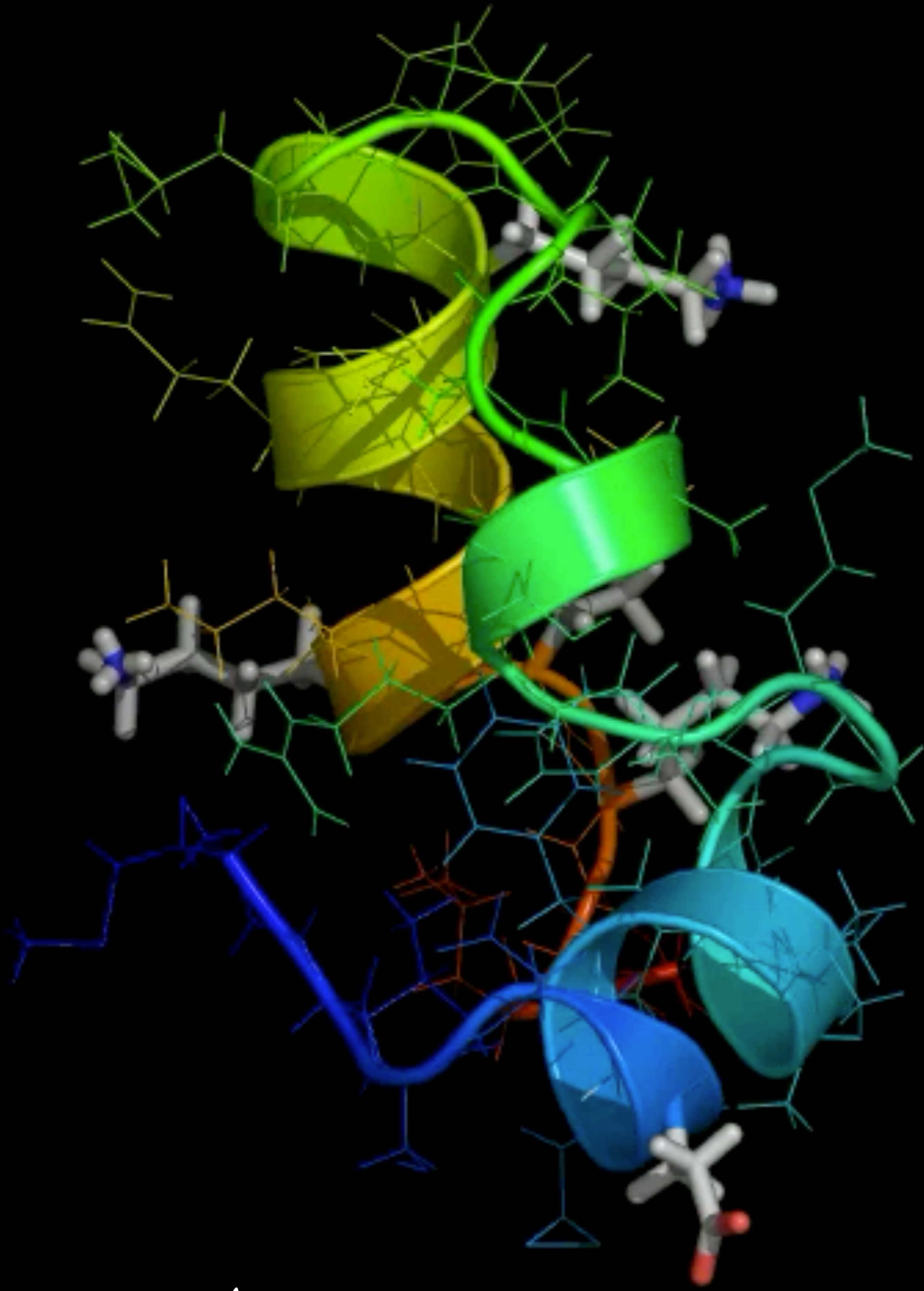
Background: This study investigated whether large fluctuations in food availability during grandparents' early development influenced grandchildren's cardiovascular mortality. We reported earlier that changes in availability of food - from good to poor or from poor to good - during intrauterine development was followed by a double risk of sudden death as an adult, and that mortality rate can be associated with ancestors´ childhood availability of food. We have now studied transgenerational responses (TGR) to sharp differences of harvest between two consecutive years´ for ancestors of 317 people in Överkalix, Sweden.

Results: The confidence intervals were very wide but we found a striking TGR. There was no response in



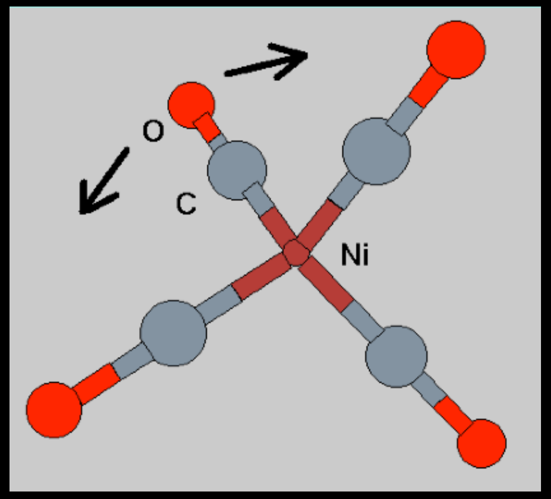


Ceci n'est pas une pipe.

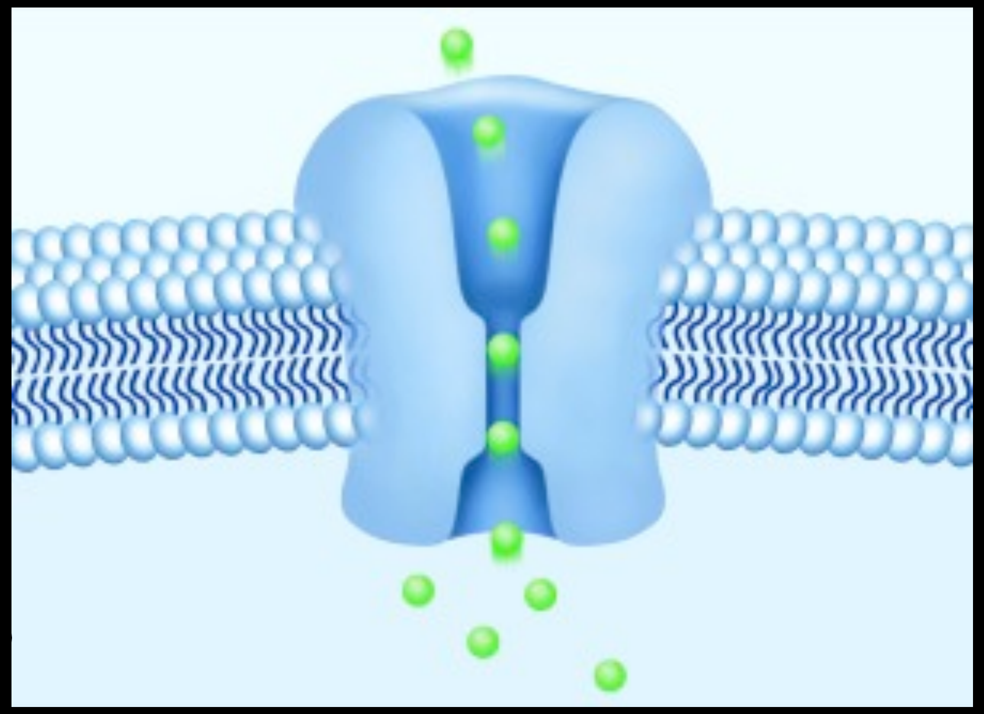


This is not a protein

$$[\frac{-\hbar^2}{2m}\nabla^2+V]\Psi=i\hbar\frac{\partial}{\partial t}\Psi$$



F=ma



Experiments

Efficient averaging

Less detail

Where we need to be

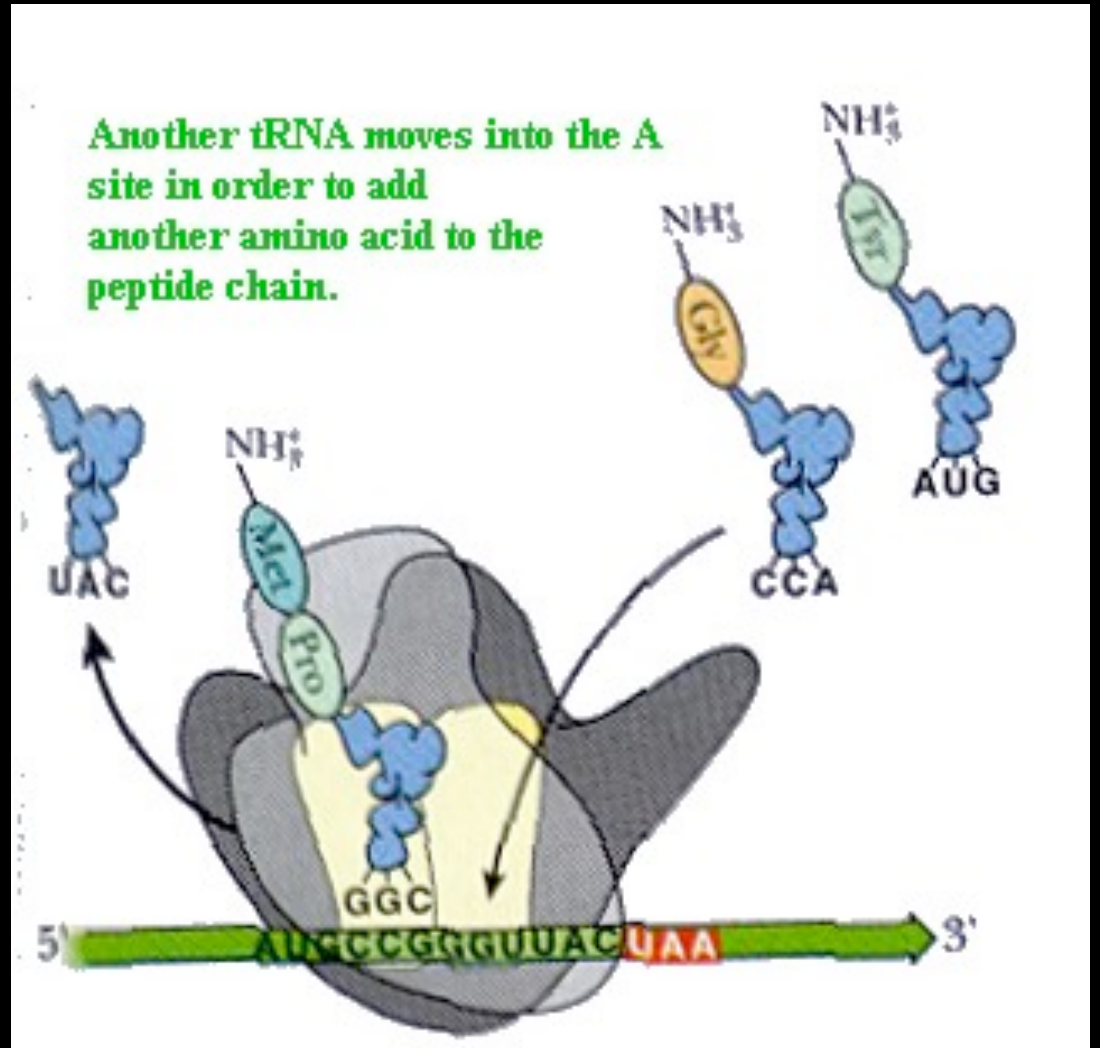
Physics | Chemistry | Biology

10^{-15} s — 10^{-12} s — 10^{-9} s — 10^{-6} s — 10^{-3} s — 10^{0} s — 10^{3} s

Simulations

Where we are

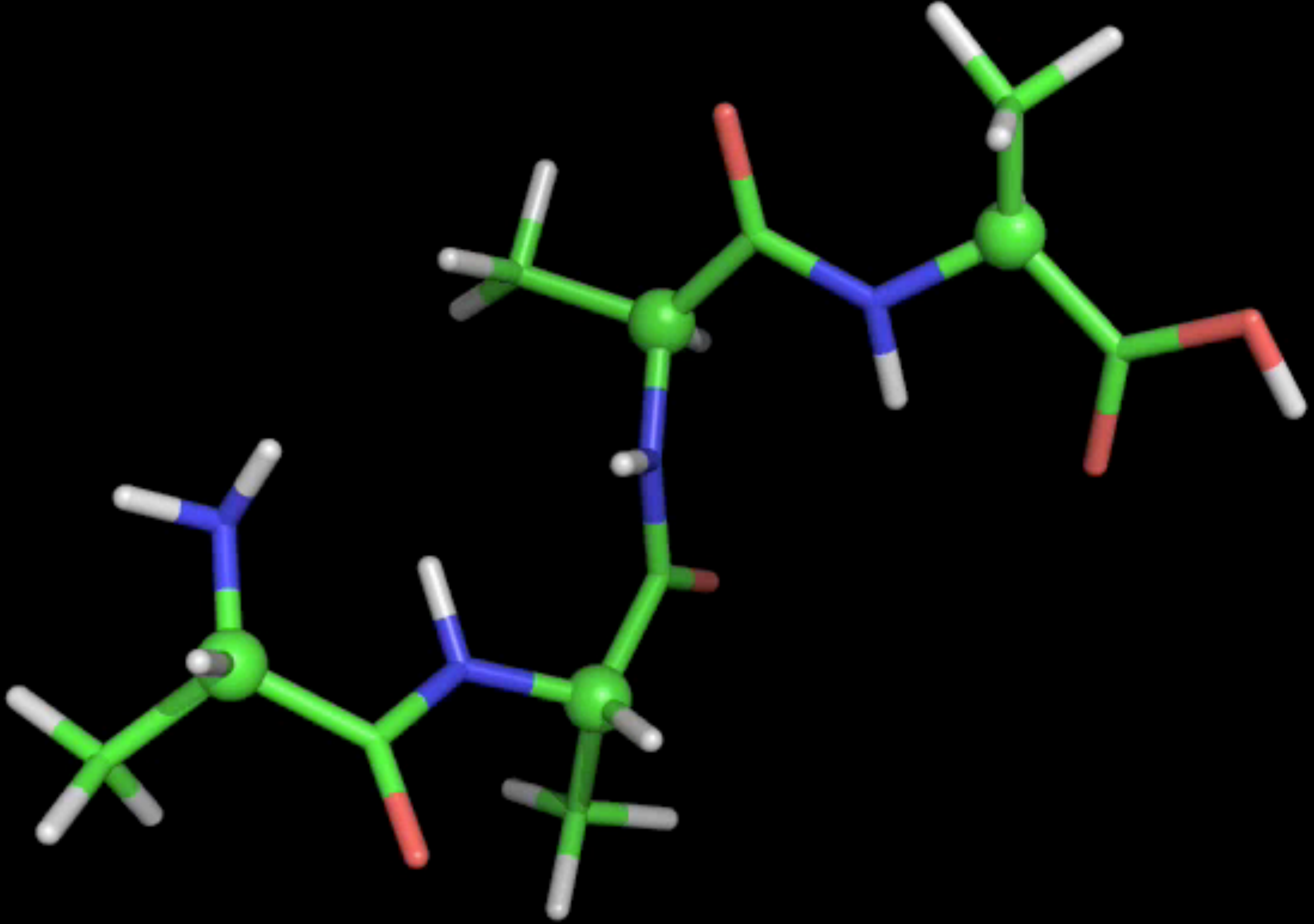
Where we want to be

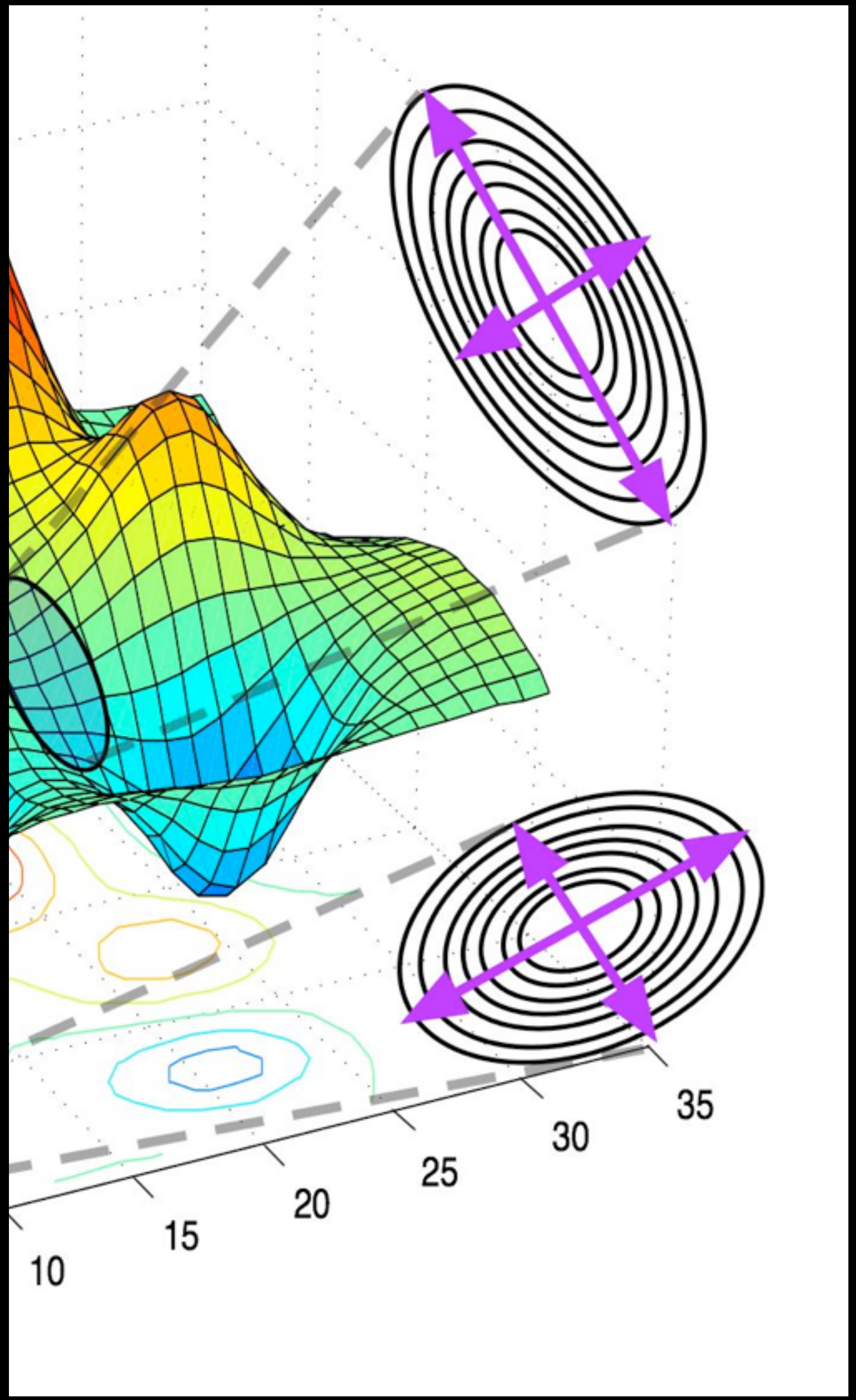

Another tRNA moves into the A site in order to add another amino acid to the peptide chain.
NH$_3^+$
Tyr
Gly
Met
Pro
UAC
AUG
CCA
GGC
5'
AUGCCGGGUUACUAA
3'

Extreme detail

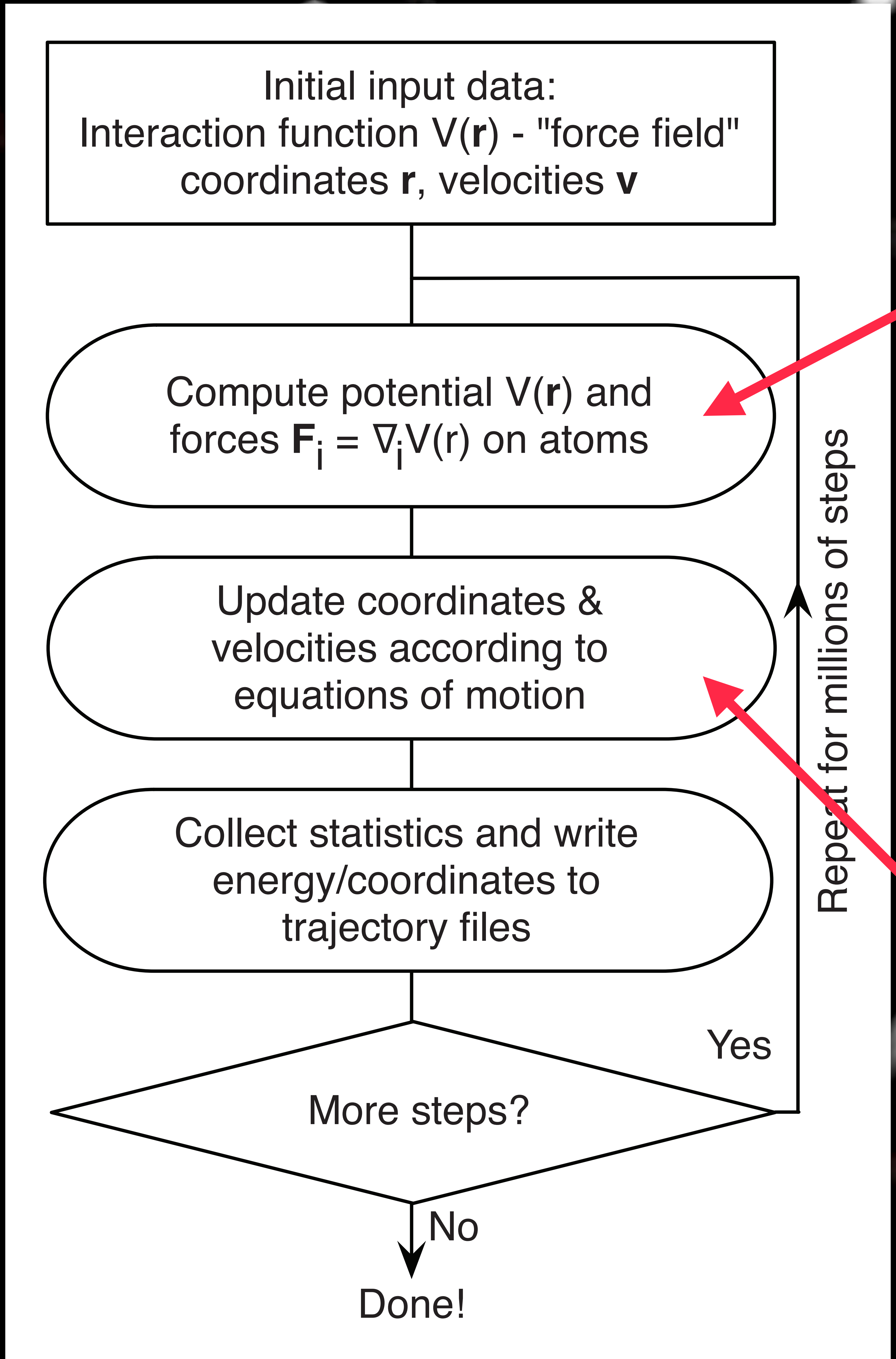
Sampling issues?

Parameter quality?




10
15
20
25
30
35

Challenge: MD is intrinsically a *sequential* problem



Initial input data:
Interaction function V(**r**) - "force field"
coordinates **r**, velocities **v**

Compute potential V(**r**) and forces $\mathbf{F}_i = \nabla_i V(r)$ on atoms

Update coordinates & velocities according to equations of motion

Collect statistics and write energy/coordinates to trajectory files

More steps?

Yes

Repeat for millions of steps

No

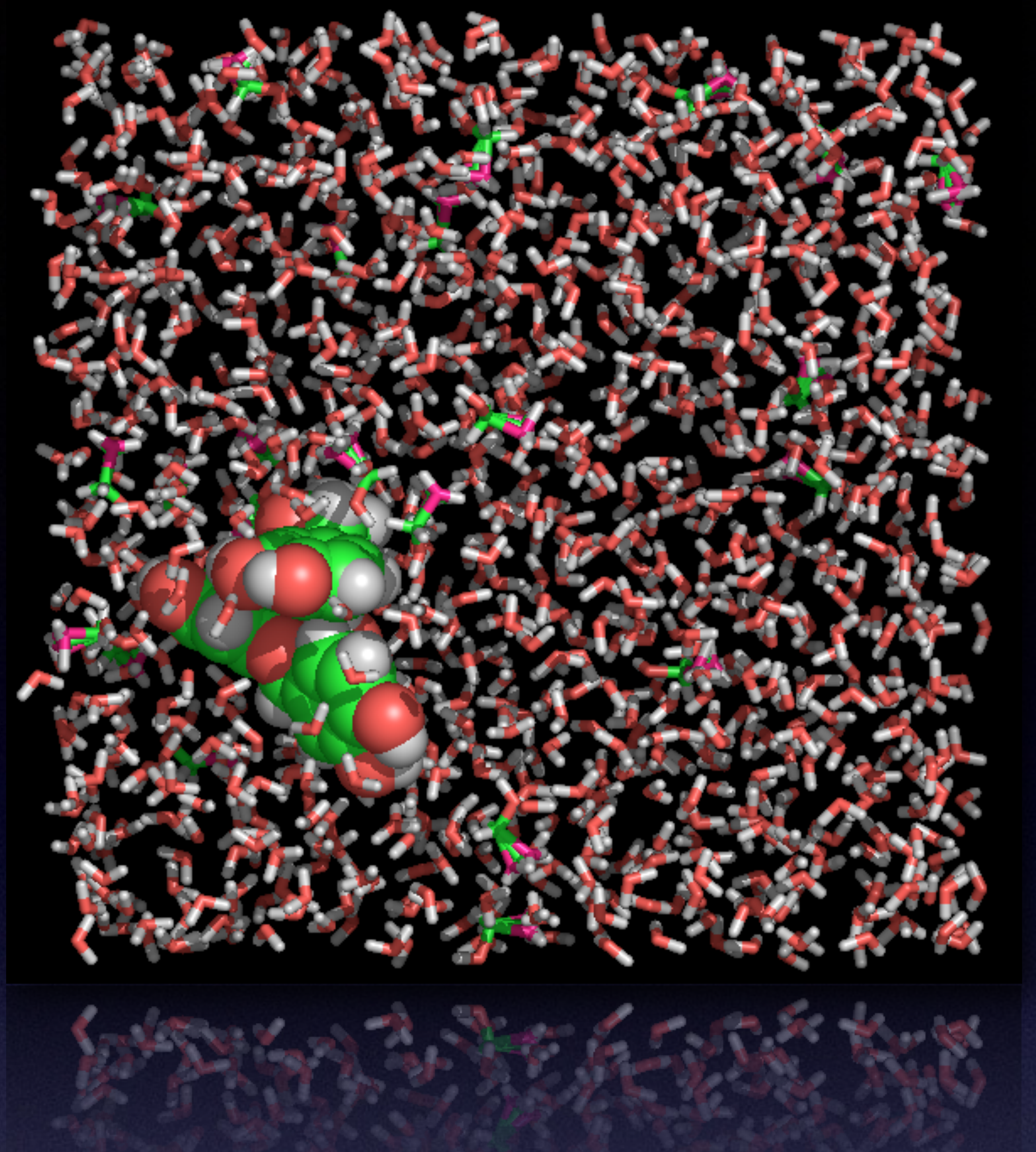
Done!

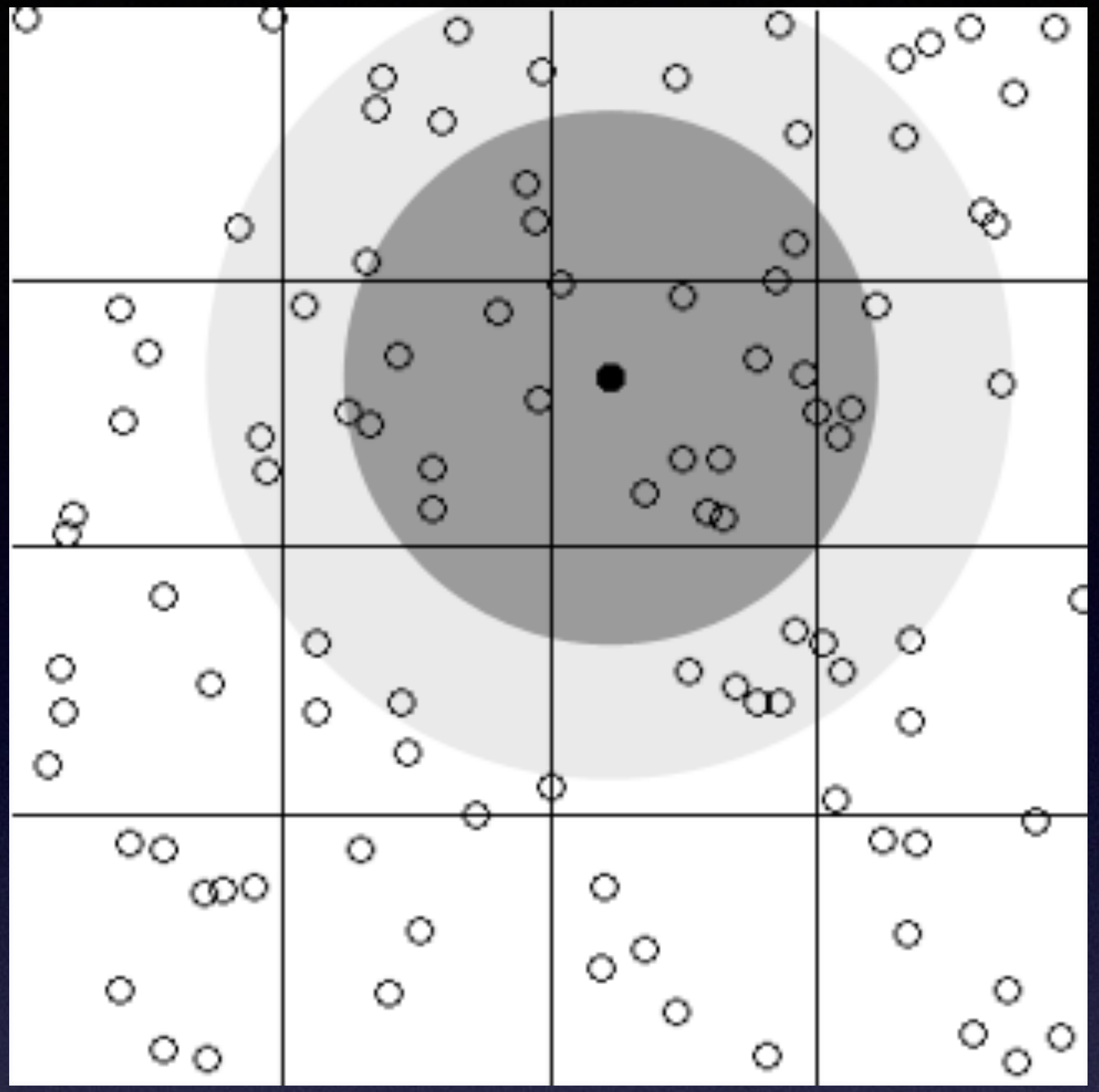
$$V(r) = \sum_{bonds} \frac{1}{2} k^b_{ij} \left(r_{ij} - r^0_{ij}\right)^2$$
$$+ \sum_{angles} \frac{1}{2} k^\theta_{ijk} \left(\theta_{ijk} - \theta^0_{ijk}\right)^2$$
$$+ \sum_{torsions} \left\{ \sum_n k_\theta \left[1 + \cos\left(n\phi - \phi_0\right)\right] \right\}$$
$$+ \sum_{impropers} k_\xi \left(\xi_{ijkl} - \xi^0_{ijkl}\right)$$
$$+ \sum_{i,j} \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}}$$
$$+ \sum_{i,j} \left[\frac{C_{12}}{r^{12}_{ij}} - \frac{C_6}{r^6_{ij}}\right]$$

Costly, because these terms involve all pairs

$$m_i \frac{\partial^2 r_i}{\partial t^2} = F_i \quad i = 1..N$$
$$F_i = -\frac{\partial V(r)}{\partial r_i}$$

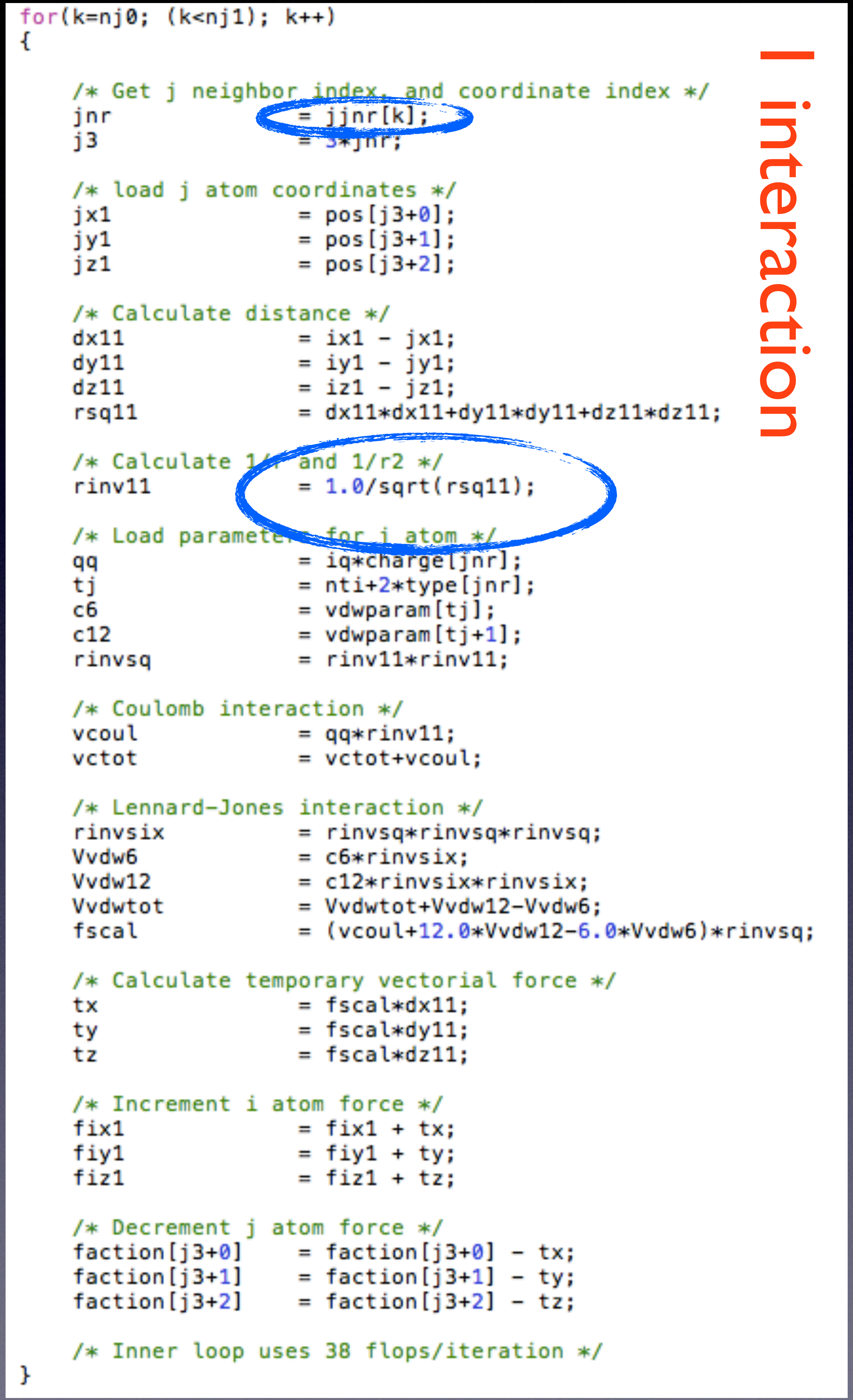
With Δt ~ 1fs and μs to s timescales of interest, we need 10^9-10^{15} steps.





The challenge:

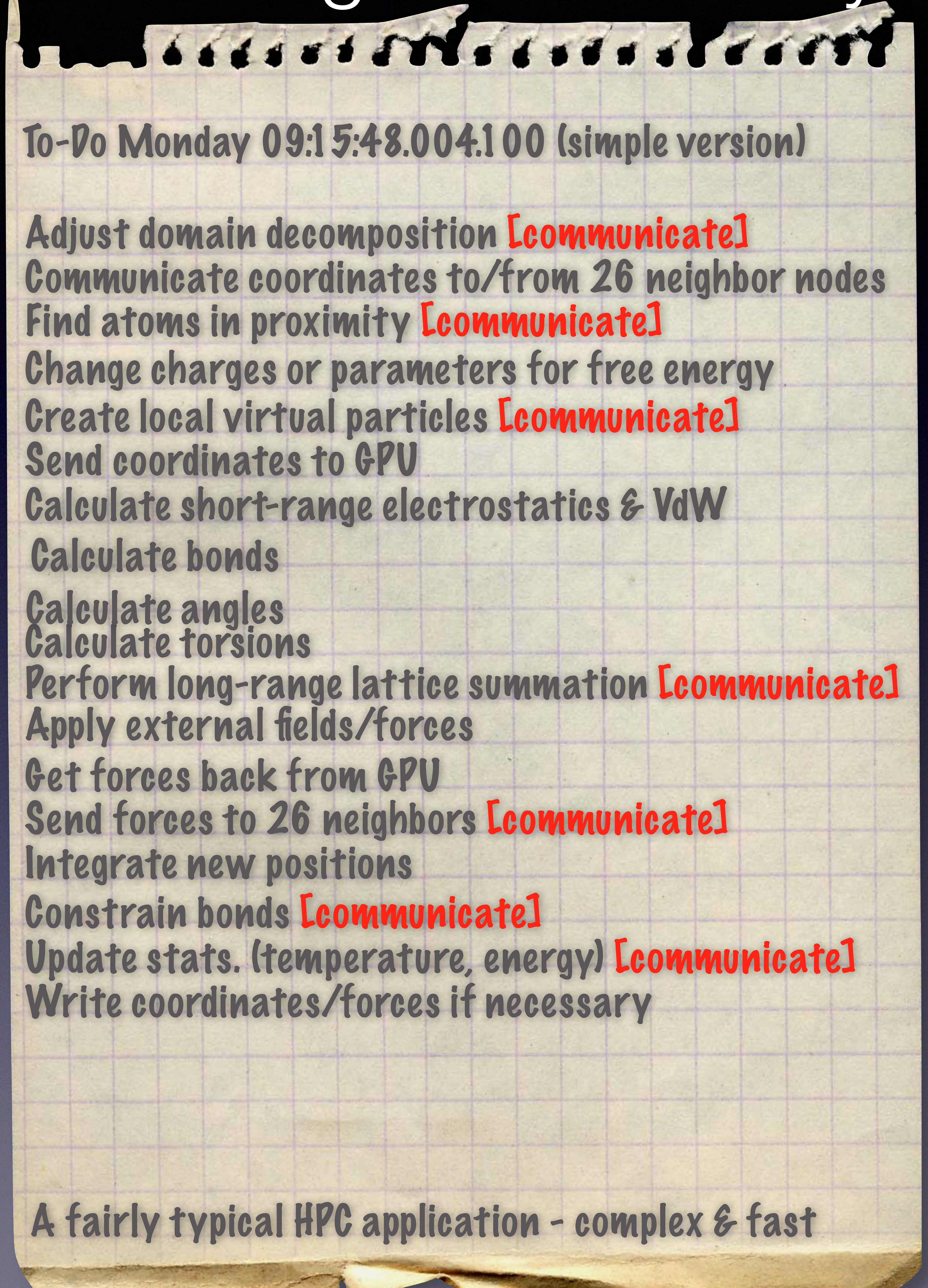
- ~100,000 atoms
- Each has ~500 neighbors
  - Maintain a list of them, update ea 10 steps
- ~50M interactions/step
- ~2B FLOPS per step
- ~1ms real time per step



```
for(k=nj0; (k<nj1); k++)
{

    /* Get j neighbor index, and coordinate index */
    jnr                = jjnr[k];
    j3                 = 3*jnr;

    /* load j atom coordinates */
    jx1                = pos[j3+0];
    jy1                = pos[j3+1];
    jz1                = pos[j3+2];

    /* Calculate distance */
    dx11               = ix1 - jx1;
    dy11               = iy1 - jy1;
    dz11               = iz1 - jz1;
    rsq11              = dx11*dx11+dy11*dy11+dz11*dz11;

    /* Calculate 1/r and 1/r2 */
    rinv11             = 1.0/sqrt(rsq11);

    /* Load parameters for j atom */
    qq                 = iq*charge[jnr];
    tj                 = nti+2*type[jnr];
    c6                 = vdwparam[tj];
    c12                = vdwparam[tj+1];
    rinvsq             = rinv11*rinv11;

    /* Coulomb interaction */
    vcoul              = qq*rinv11;
    vctot              = vctot+vcoul;

    /* Lennard-Jones interaction */
    rinvsix            = rinvsq*rinvsq*rinvsq;
    Vvdw6              = c6*rinvsix;
    Vvdw12             = c12*rinvsix*rinvsix;
    Vvdwtot            = Vvdwtot+Vvdw12-Vvdw6;
    fscal              = (vcoul+12.0*Vvdw12-6.0*Vvdw6)*rinvsq;

    /* Calculate temporary vectorial force */
    tx                 = fscal*dx11;
    ty                 = fscal*dy11;
    tz                 = fscal*dz11;

    /* Increment i atom force */
    fix1               = fix1 + tx;
    fiy1               = fiy1 + ty;
    fiz1               = fiz1 + tz;

    /* Decrement j atom force */
    faction[j3+0]      = faction[j3+0] - tx;
    faction[j3+1]      = faction[j3+1] - ty;
    faction[j3+2]      = faction[j3+2] - tz;

    /* Inner loop uses 38 flops/iteration */
}
```

1 interaction

The things we do every ~100 µs

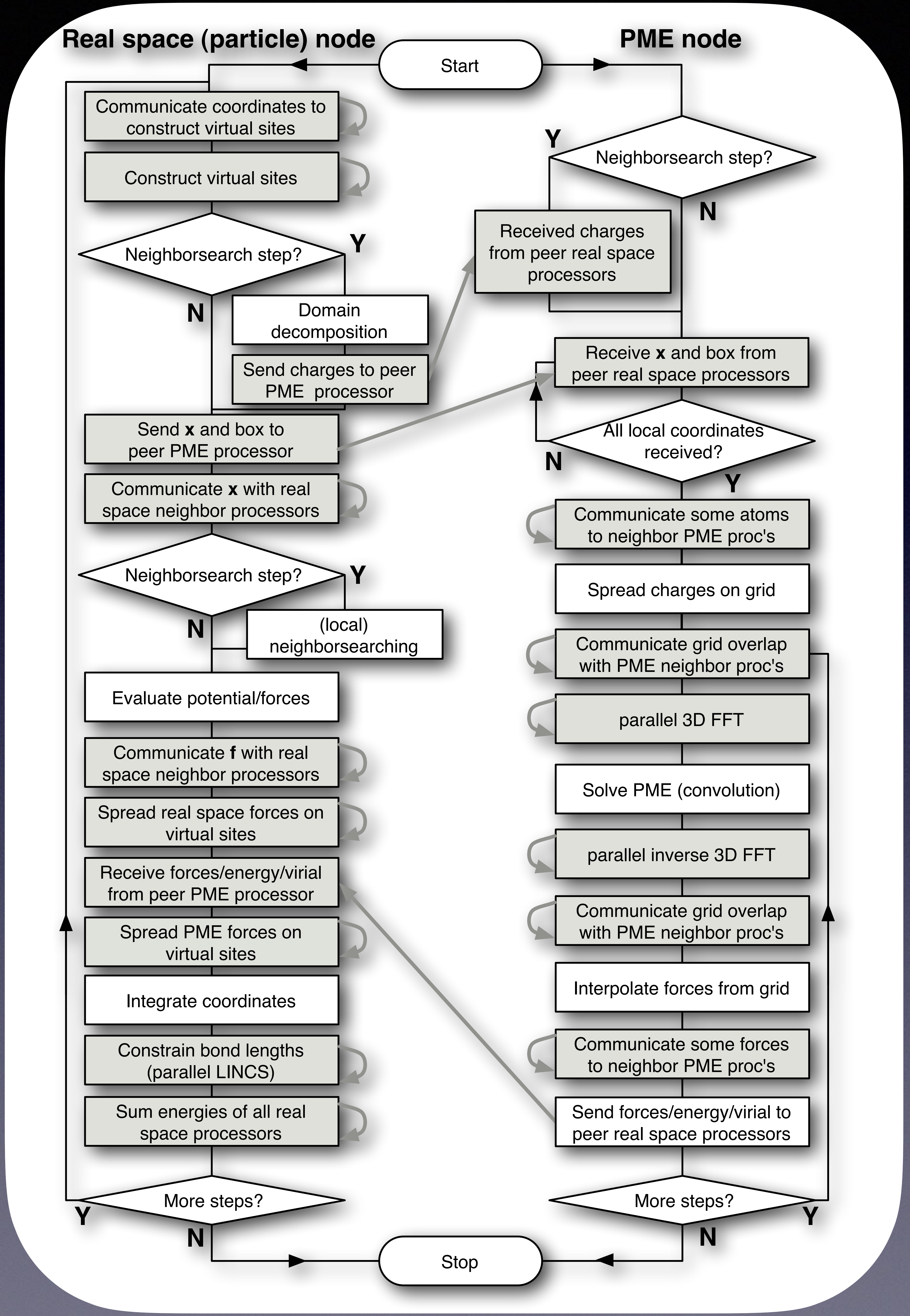


To-Do Monday 09:15:48.004100 (simple version)

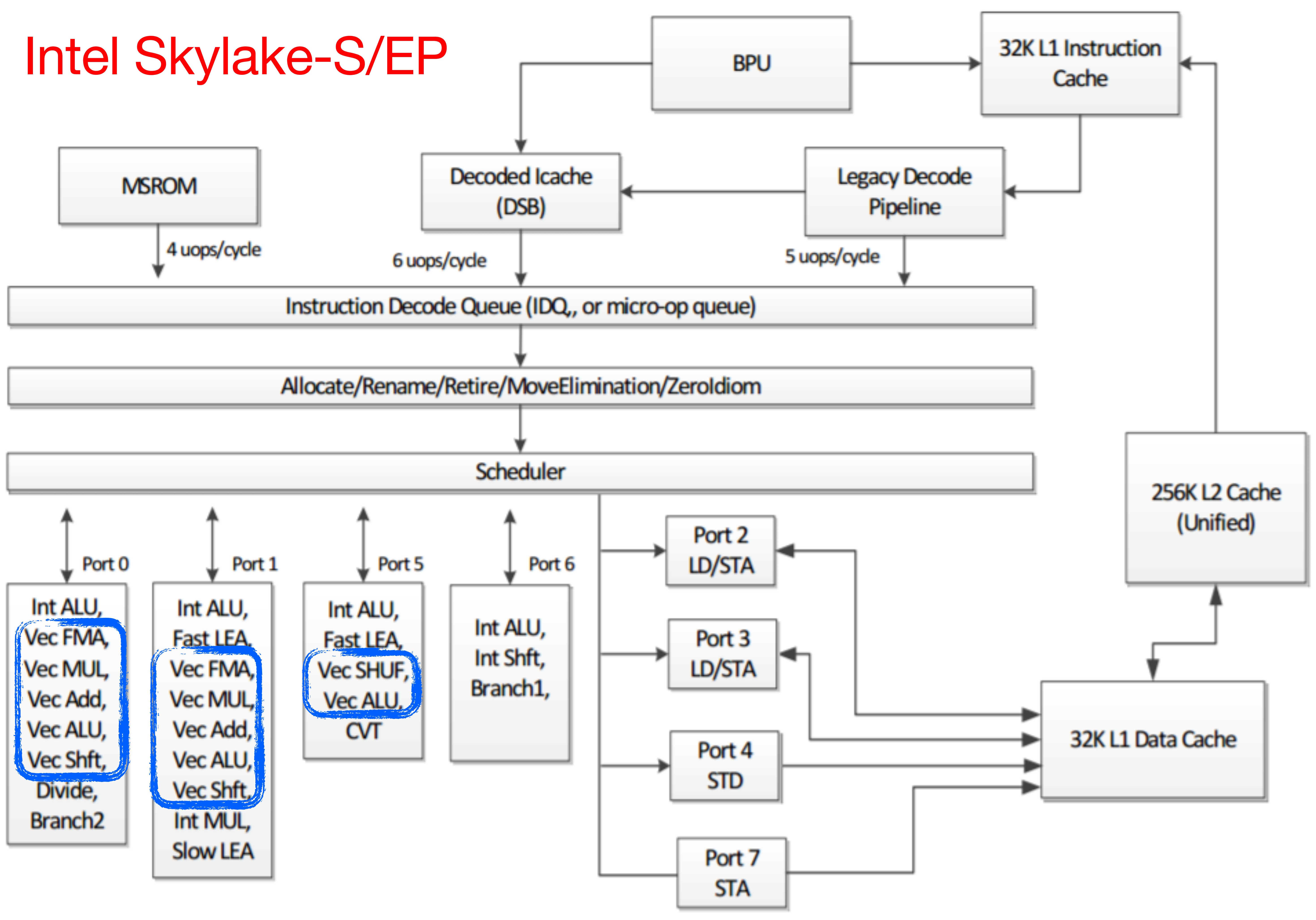
Adjust domain decomposition [communicate]
Communicate coordinates to/from 26 neighbor nodes
Find atoms in proximity [communicate]
Change charges or parameters for free energy
Create local virtual particles [communicate]
Send coordinates to GPU
Calculate short-range electrostatics & VdW
Calculate bonds
Calculate angles
Calculate torsions
Perform long-range lattice summation [communicate]
Apply external fields/forces
Get forces back from GPU
Send forces to 26 neighbors [communicate]
Integrate new positions
Constrain bonds [communicate]
Update stats. (temperature, energy) [communicate]
Write coordinates/forces if necessary

A fairly typical HPC application - complex & fast

Every arrow is communication


Real space (particle) node
Start
PME node
Communicate coordinates to construct virtual sites
Construct virtual sites
Neighborsearch step?
Y
N
Domain decomposition
Send charges to peer PME processor
Send x and box to peer PME processor
Communicate x with real space neighbor processors
Neighborsearch step?
Y
N
(local) neighborsearching
Evaluate potential/forces
Communicate f with real space neighbor processors
Spread real space forces on virtual sites
Receive forces/energy/virial from peer PME processor
Spread PME forces on virtual sites
Integrate coordinates
Constrain bond lengths (parallel LINCS)
Sum energies of all real space processors
More steps?
Y
N
Stop
Neighborsearch step?
Y
N
Received charges from peer real space processors
Receive x and box from peer real space processors
All local coordinates received?
N
Y
Communicate some atoms to neighbor PME proc's
Spread charges on grid
Communicate grid overlap with PME neighbor proc's
parallel 3D FFT
Solve PME (convolution)
parallel inverse 3D FFT
Communicate grid overlap with PME neighbor proc's
Interpolate forces from grid
Communicate some forces to neighbor PME proc's
Send forces/energy/virial to peer real space processors
More steps?
N
Y

What does a modern CPU core look like?


Intel Skylake-S/EP
BPU
32K L1 Instruction Cache
MSROM
Decoded Icache (DSB)
Legacy Decode Pipeline
4 uops/cycle
6 uops/cycle
5 uops/cycle
Instruction Decode Queue (IDQ,, or micro-op queue)
Allocate/Rename/Retire/MoveElimination/ZeroIdiom
Scheduler
256K L2 Cache (Unified)
Port 0
Port 1
Port 5
Port 6
Int ALU, Vec FMA, Vec MUL, Vec Add, Vec ALU, Vec Shft, Divide, Branch2
Int ALU, Fast LEA, Vec FMA, Vec MUL, Vec Add, Vec ALU, Vec Shft, Int MUL, Slow LEA
Int ALU, Fast LEA, Vec SHUF, Vec ALU, CVT
Int ALU, Int Shft, Branch1,
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
32K L1 Data Cache

SIMD: 128, 256, or 512 bit vectors

5 µops/core each cycle
SIMD + FMA: 64 flops @ single prec.

Up to 32 cores per chip

2 sockets per node

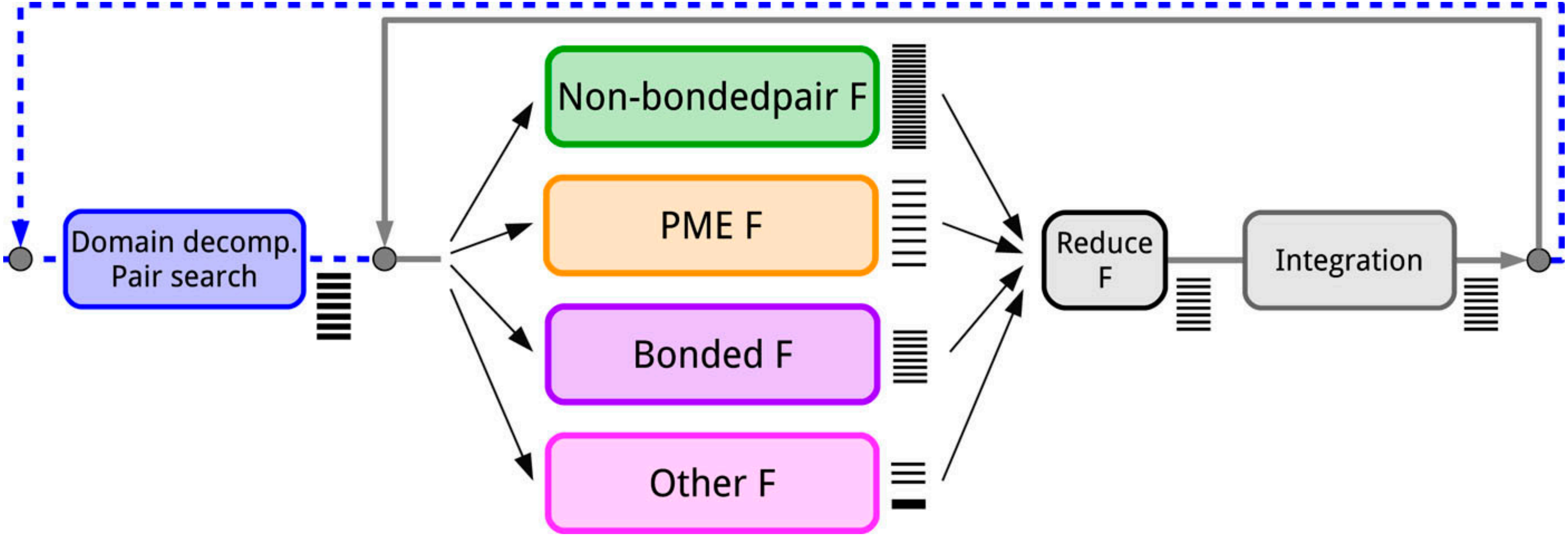
Theoretically:
320 instructions or 4096 flops per *cycle* on each node.

latency is ~4 cycles on Skylake. You need *256* independent FMA flops (128 FMA operations) to saturate just a single core

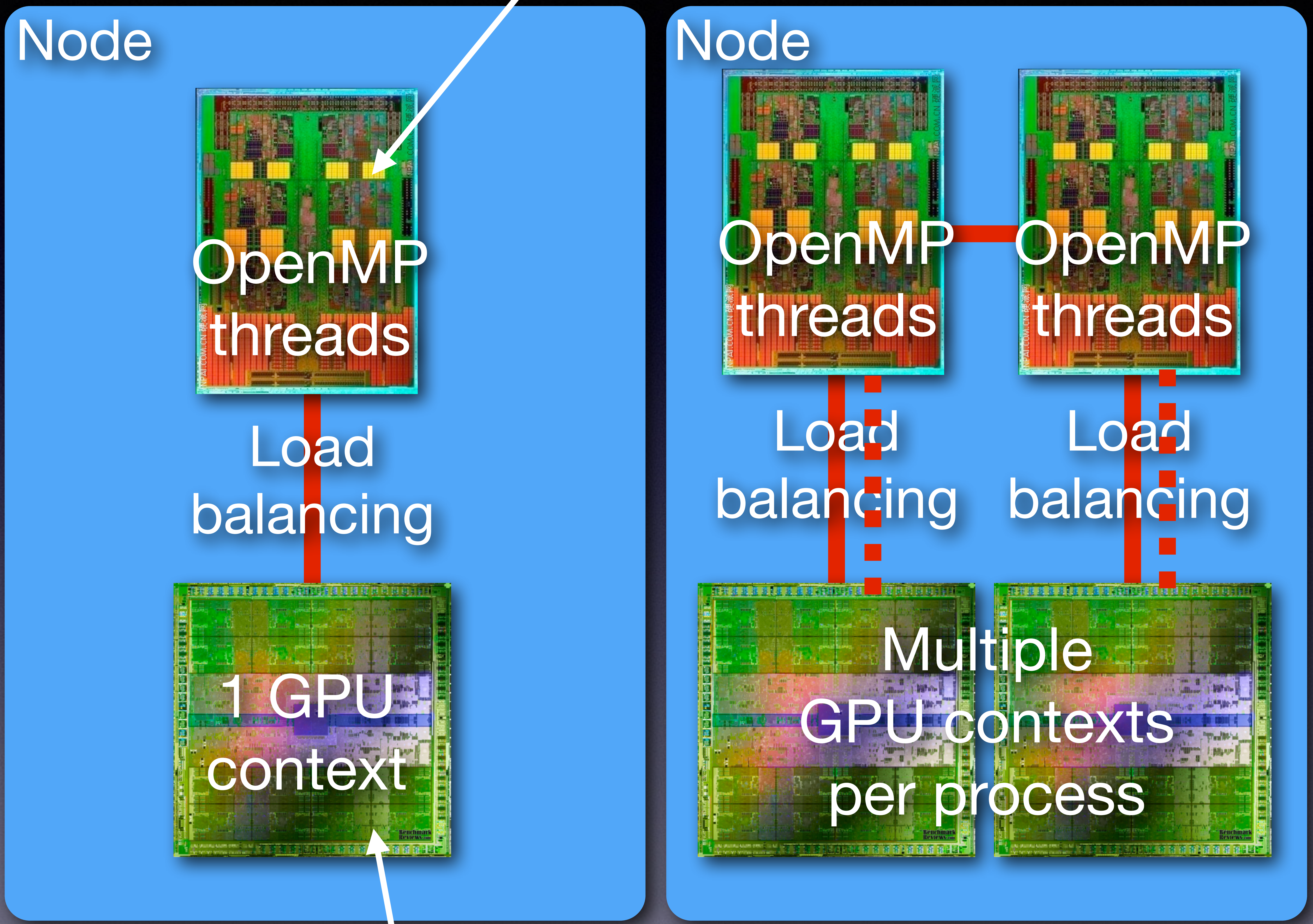
PCI Express 3.0 Host Interface
GigaThread Engine
HBM2
Memory Controller
GPC
TPC
SM
L2 Cache
High-Speed Hub
NVLink
NVLink
NVLink
NVLink
NVLink
NVLink

Acceleration Approaches

	GPU libraries	OpenAcc	Pure CUDA	Heterogeneous CPU/GPU
Initial effort / Expertise req.	🟩	🟨	🟧	🟥
Generality / Portability	🟩	🟩	🟥	🟩
Performance	🟩–🟥	🟨	🟨	🟩
Code maintainability	🟩	🟩	🟥	🟨
	Works if your code offloads to libraries	Always works, but success depends on you & compiler	Lots of work, assumes impl. can run entirely on GPU	Even more work, less CUDA, can use both CPU & GPU


Domain decomp.
Pair search
Non-bondedpair F
PME F
Bonded F
Other F
Reduce
F
Integration

Explicit SIMD instructions on CPUs & Xeon Phi; each instruction does up to 32 flops



Node

OpenMP threads

Load balancing

1 GPU context

Node

OpenMP threads

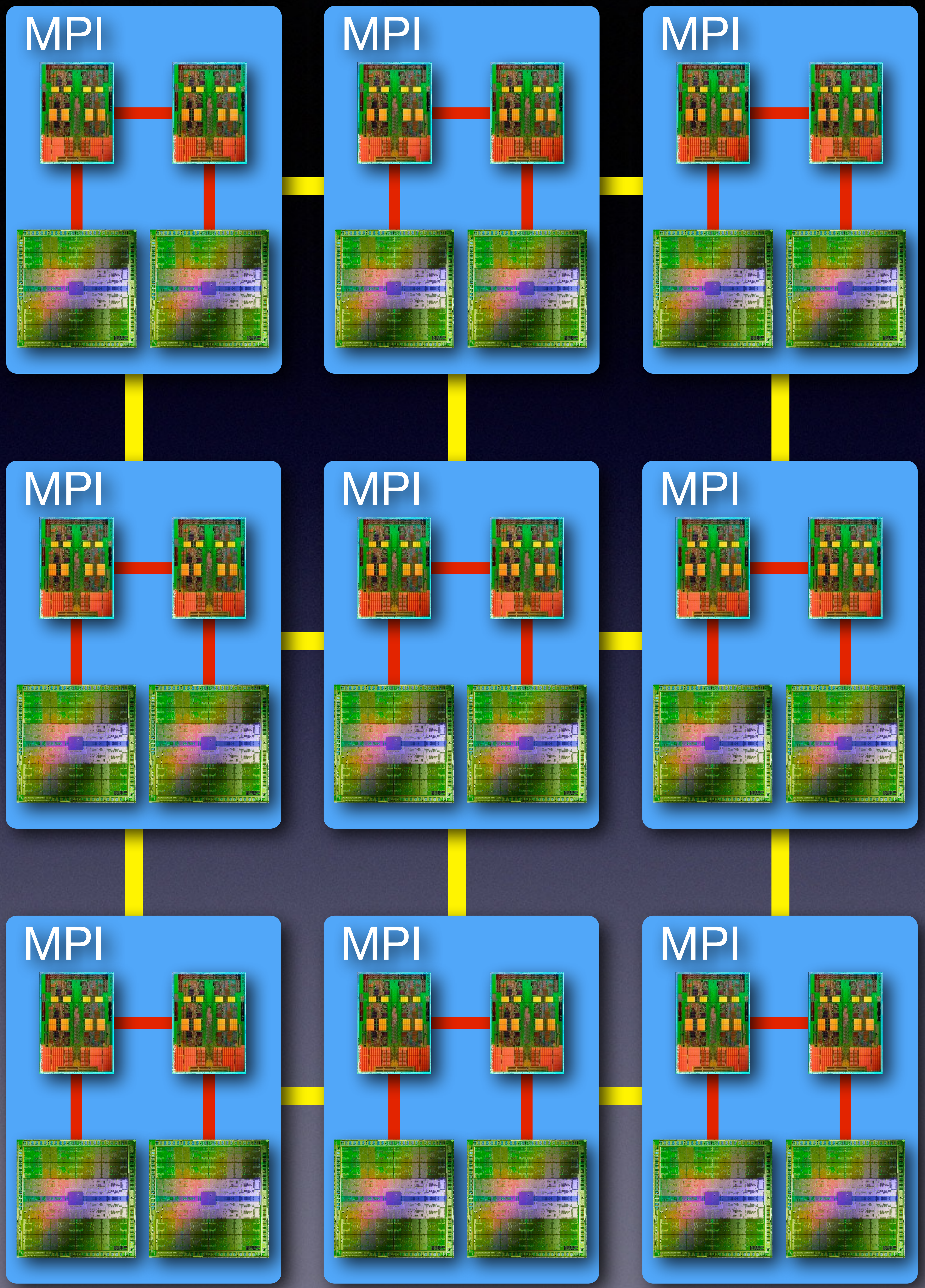
OpenMP threads

Load balancing

Load balancing

Multiple GPU contexts per process

CUDA kernels on NVIDIA GPUs, OpenCL for AMD/Intel GPUs



MPI

MPI

MPI

MPI

MPI

MPI

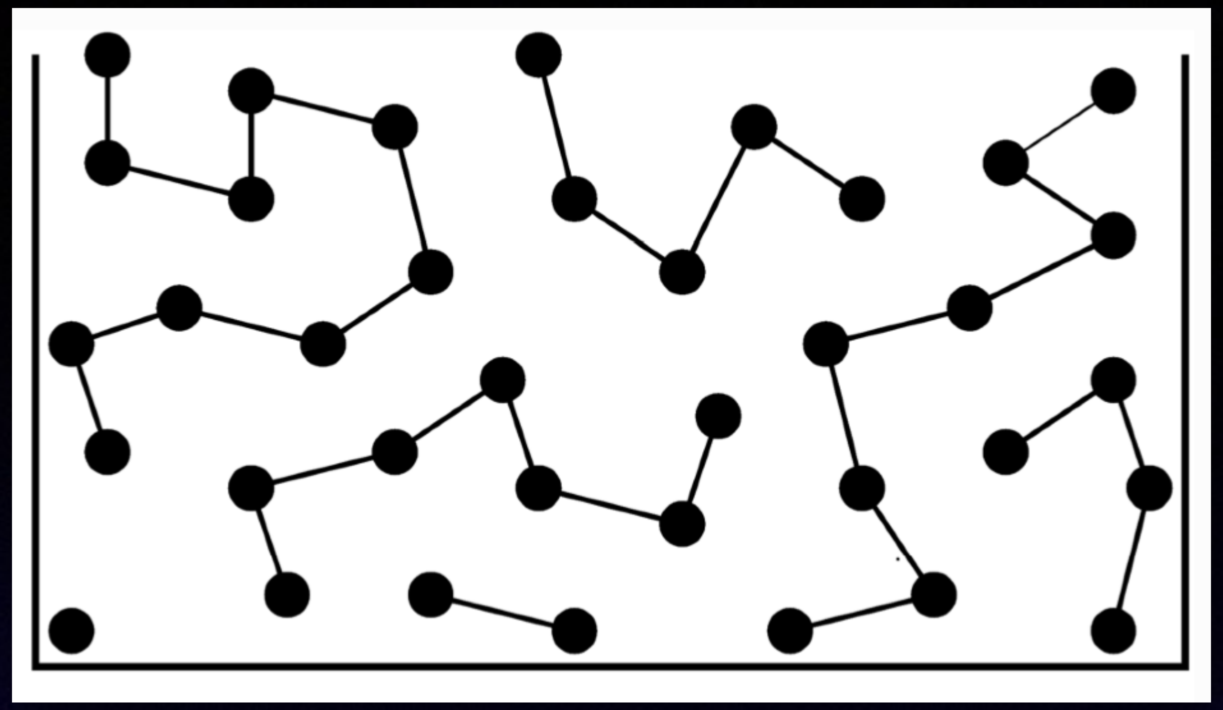
MPI

MPI

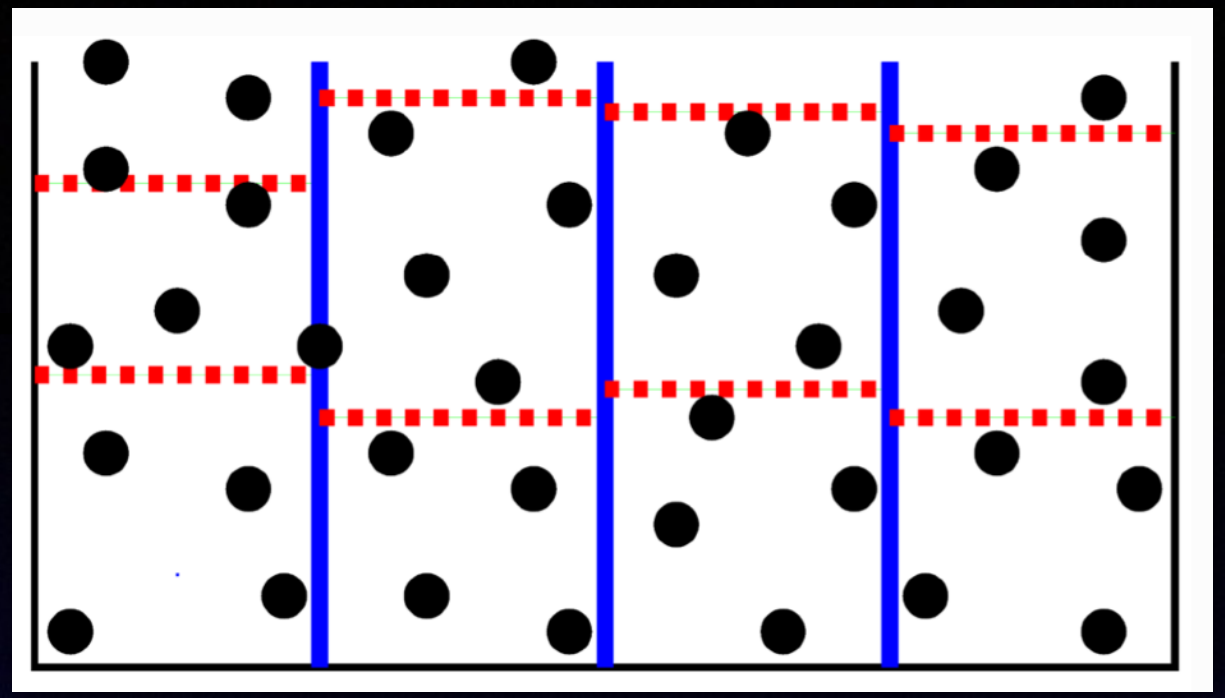
MPI

A
CPU
Pair Search
Bonded F
Non-bonded F
PME
Other Forces
Reduce Forces
Integration, Constraints
B
CPU
GPU
C
CPU
GPU
D
CPU
GPU
E
CPU
GPU

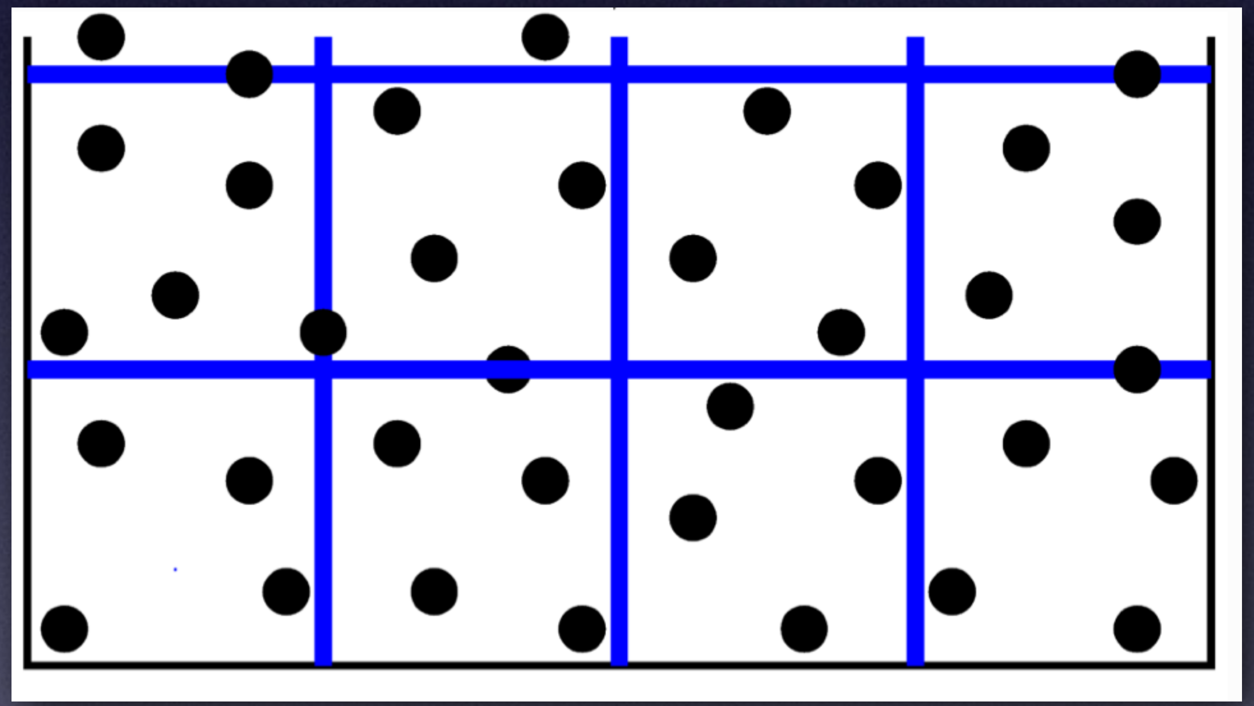
From neighborlists to cluster proximity lists: Revisit algorithms



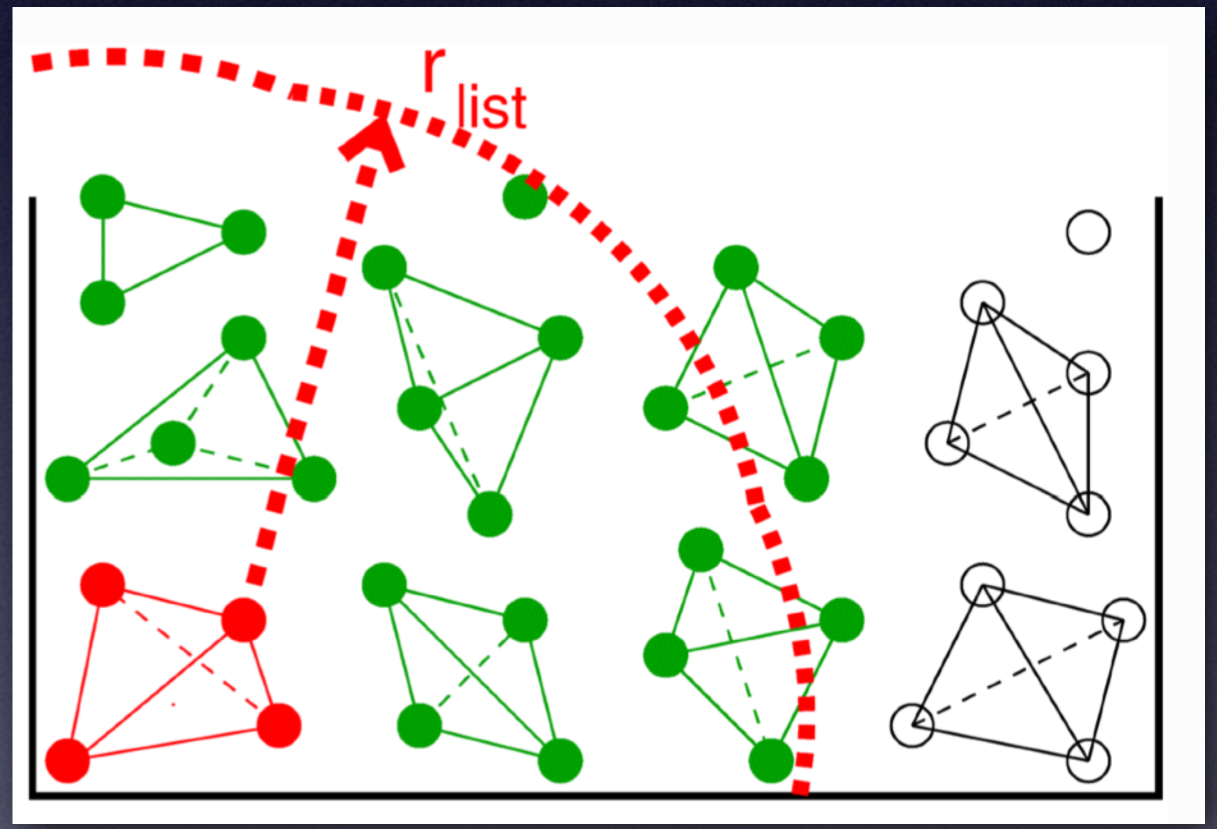
x,y grid
z sort
z bin



x,y,z
gridding



Cluster pairlist



Organize
as tiles with
all-vs-all
interactions:

X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X

i=3:	5	6	9	12	15	17	18	25	32	…		
i=4:	7	8	9	11	12	15	17	25	32	43	54	…
…	8	9	10	11	12	13	19	20	…			

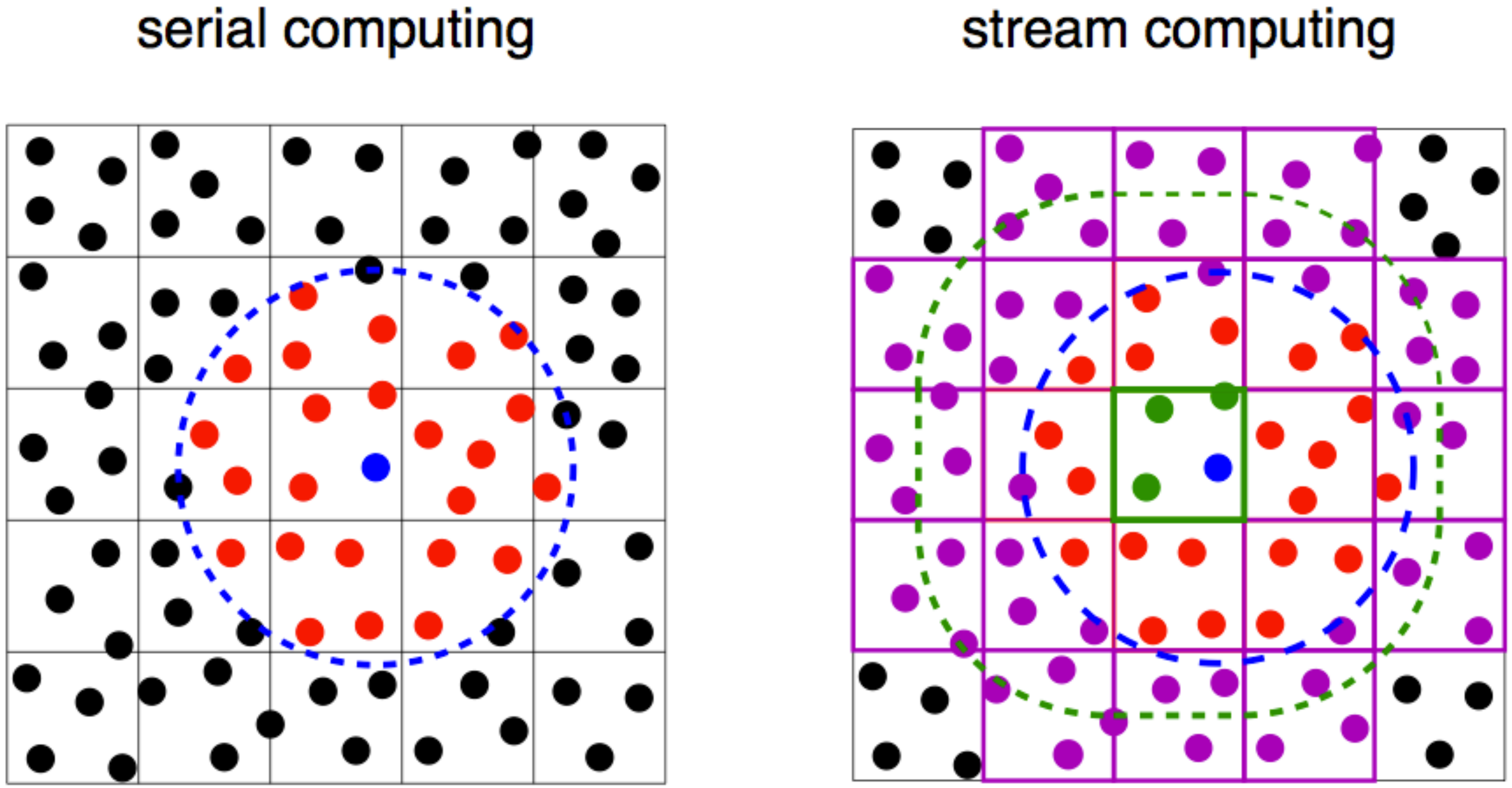
Tile interaction algorithms:
Load N atoms, compute N^2 forces

The Link-cell algorithm: Load 1 atom, calculate 1 interaction

Verlet, Phys Rev 159, 98-103 (1967)

This creates a new problem: Tiling circles is difficult

You want to calculate interactions with red neighbors



serial computing

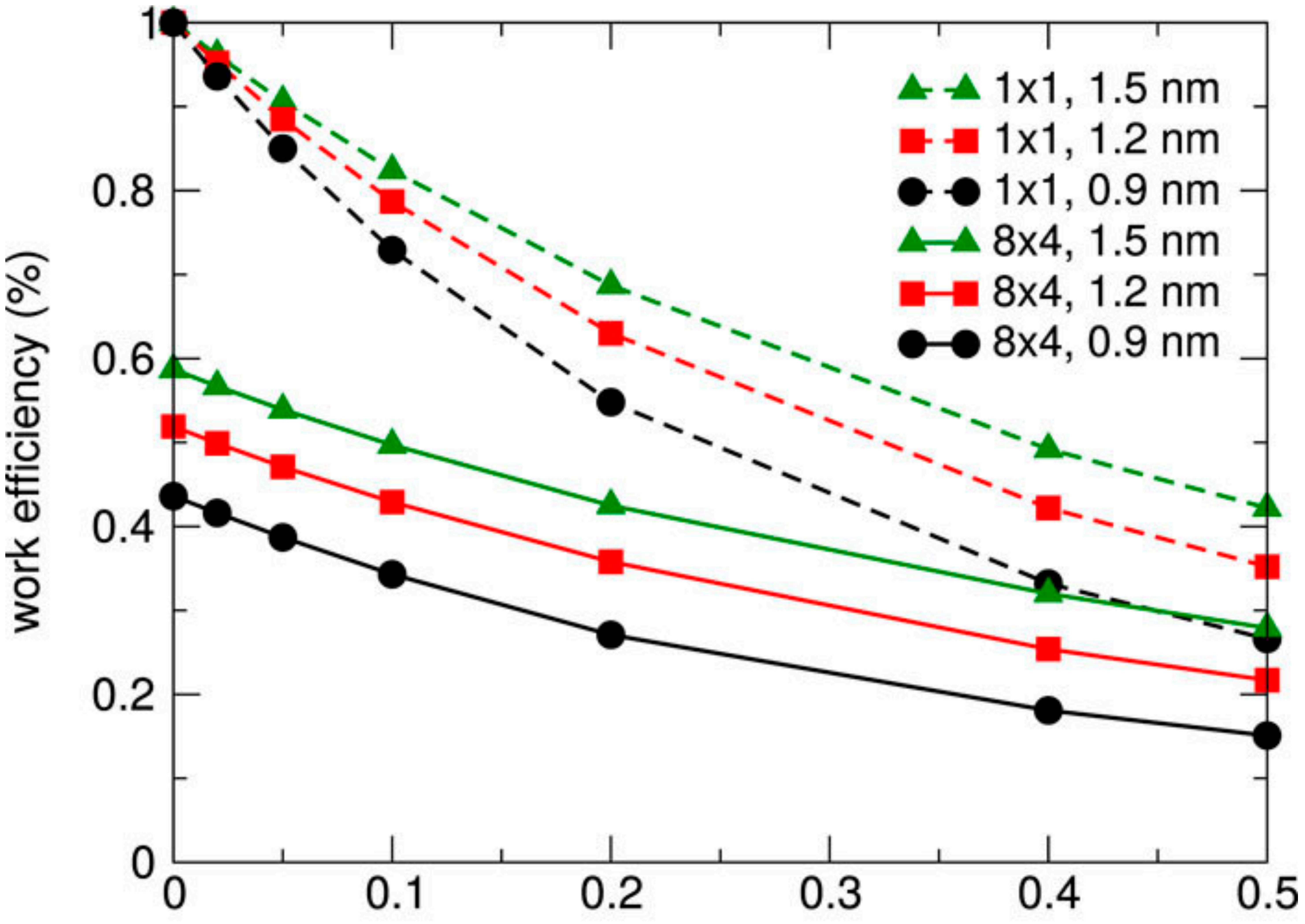
stream computing

Lots of wasted FLOPS!

- You need a lot of cubes to cover a sphere
- All interactions beyond cutoff need to be zero

Higher flop efficiency

Better GPU load efficiency



Extra space around cutoff

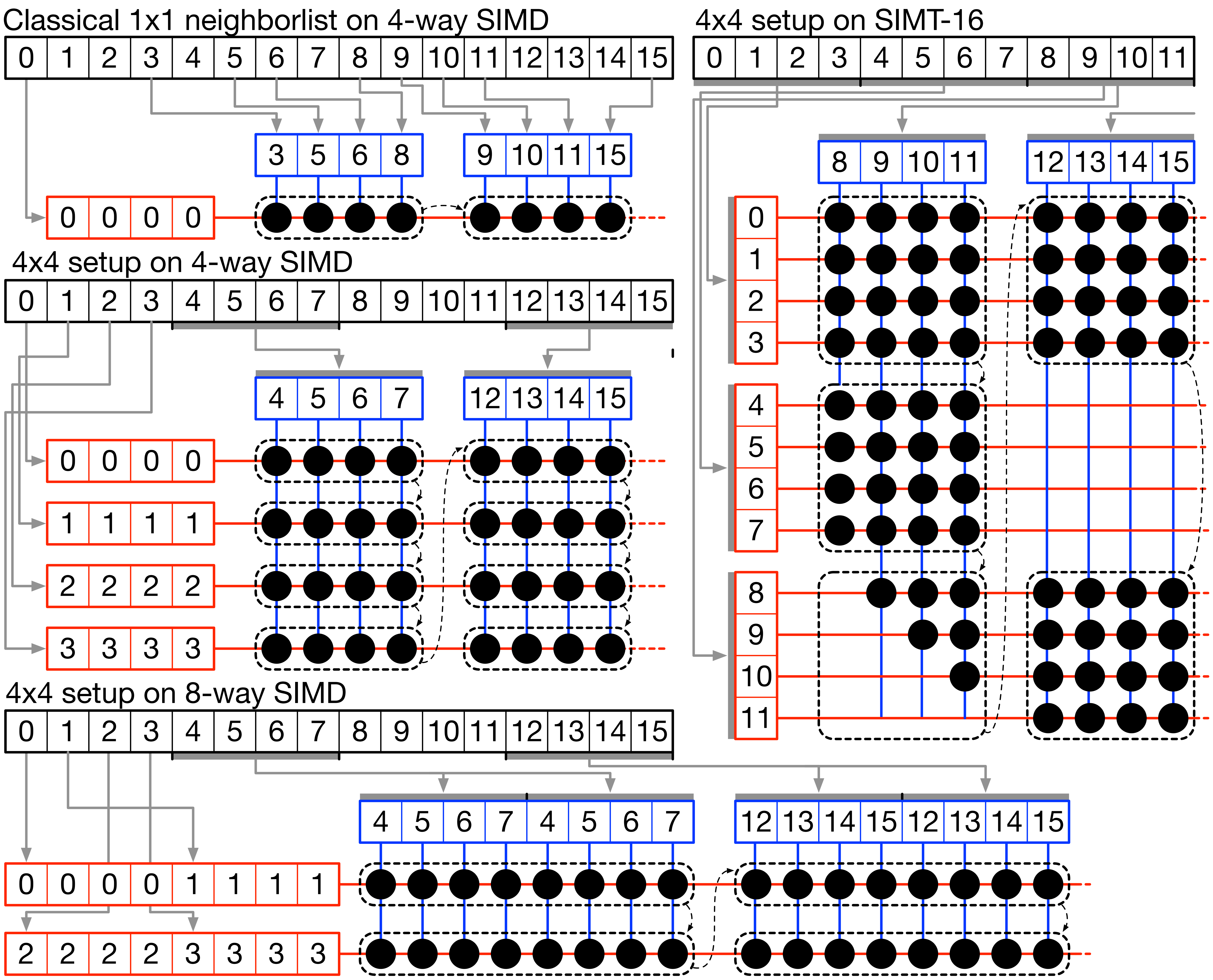
Clusters on CPUs, Superclusters on GPUs

i_1
i_1
i_2
i_3
i_4
j_m

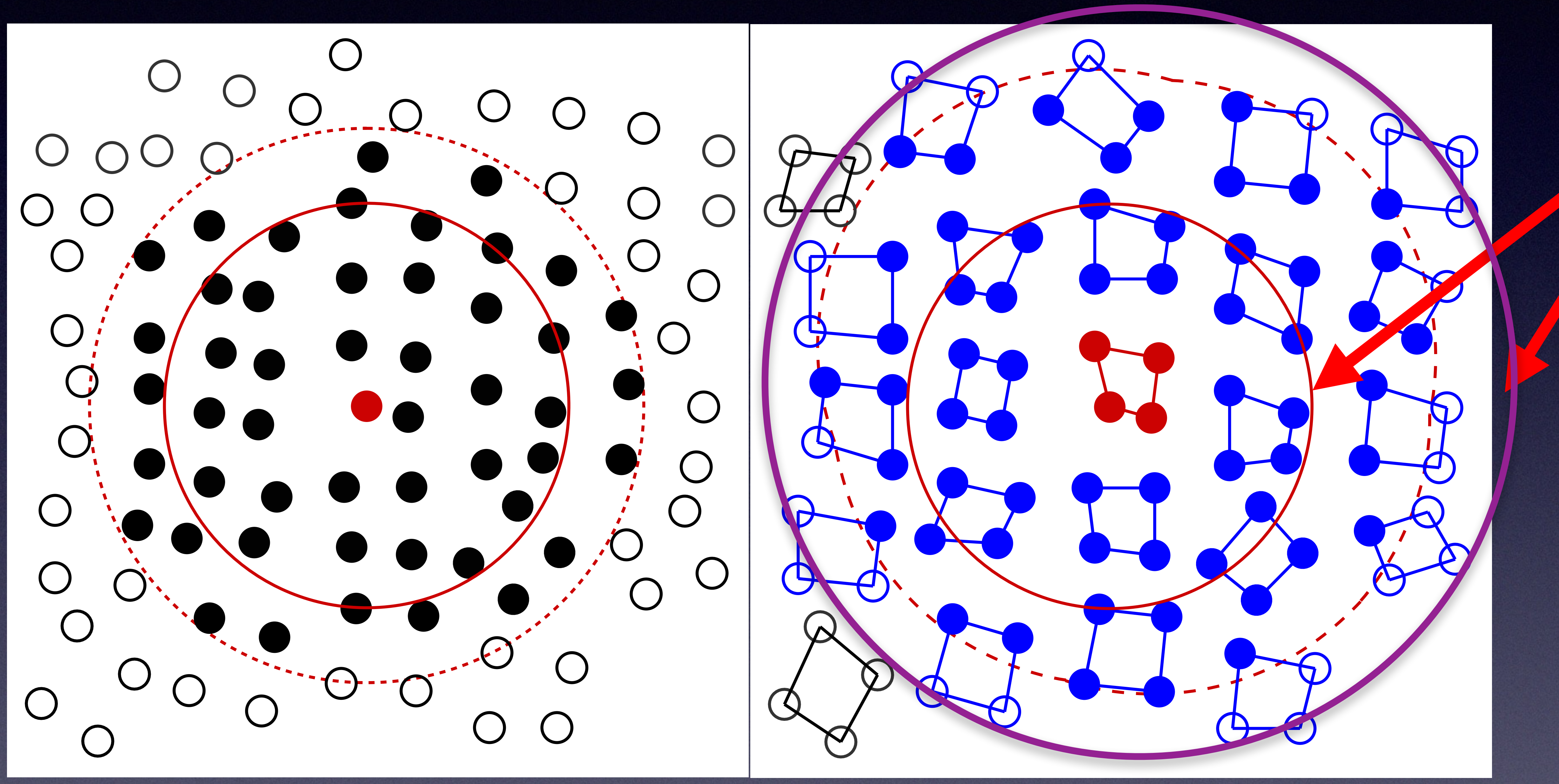
Bringing the Performace back to the CPU

Unified GPU/CPU architecture - completely portable

- CUDA
- OpenCL
- Intel MIC
- x86 SSE2
- x86 SSE4.1
- x86 AVX
- x86 AVX-128-FMA
- x86 AVX2
- x86 AVX2_128
- x86 AVX-512F
- x86 AVX-512ER
- Arm Neon
- Arm64 Asimd
- IBM QPX
- IBM VMX
- IBM VSX
- Fujitsu HPC-ACE



Atom clustering and pair list buffering

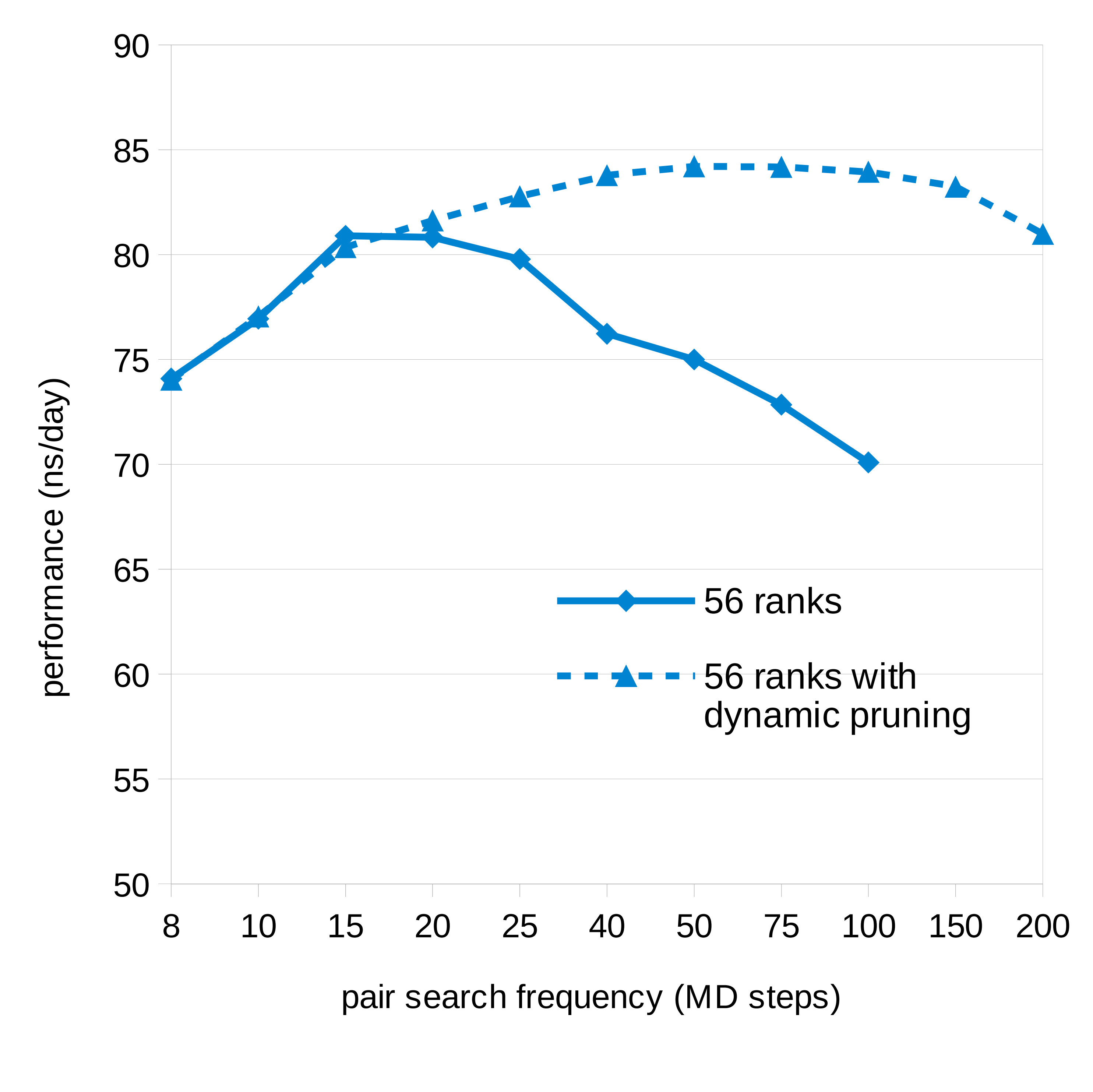


We can
Adjust the
size of this buffer

Larger buffer means more calculations, but we can update the neighbor list less frequently

New dual-pair list buffer:

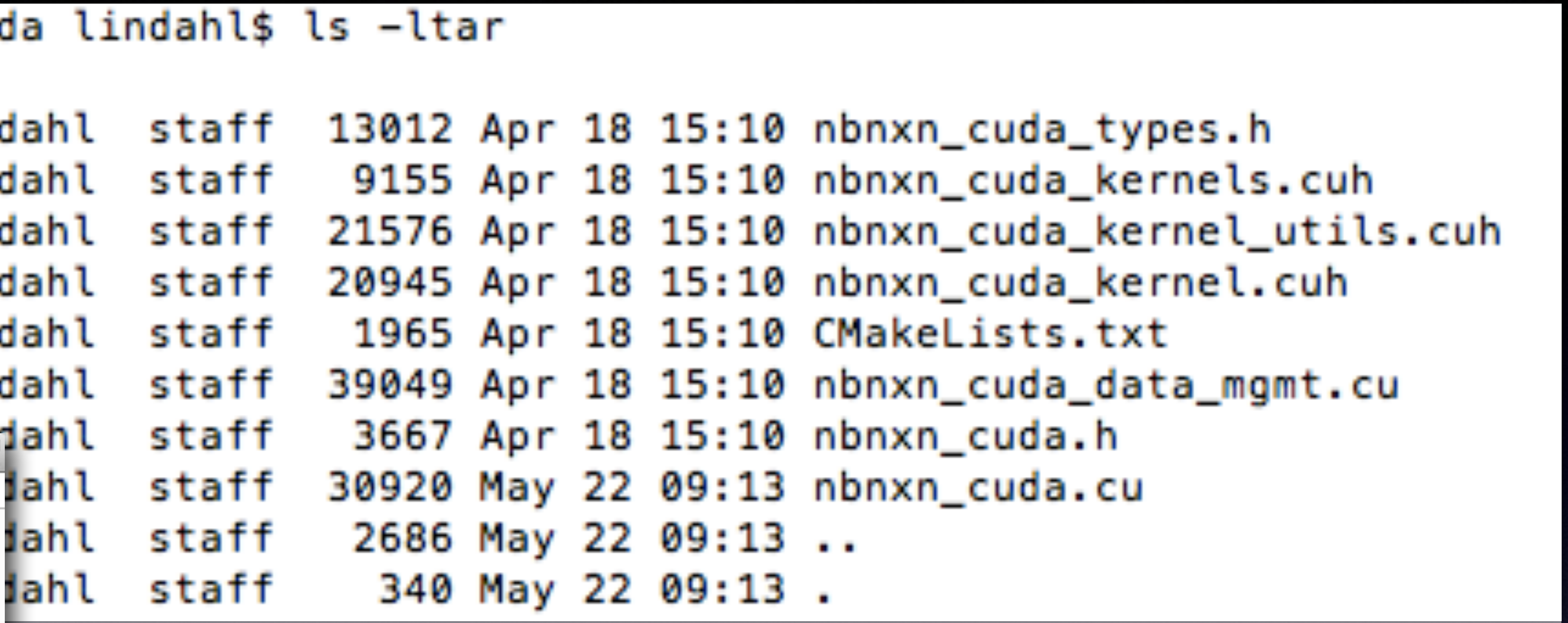
- Use very large buffers, and prune it every few steps
- reduces overhead
- less sensitive to parameters



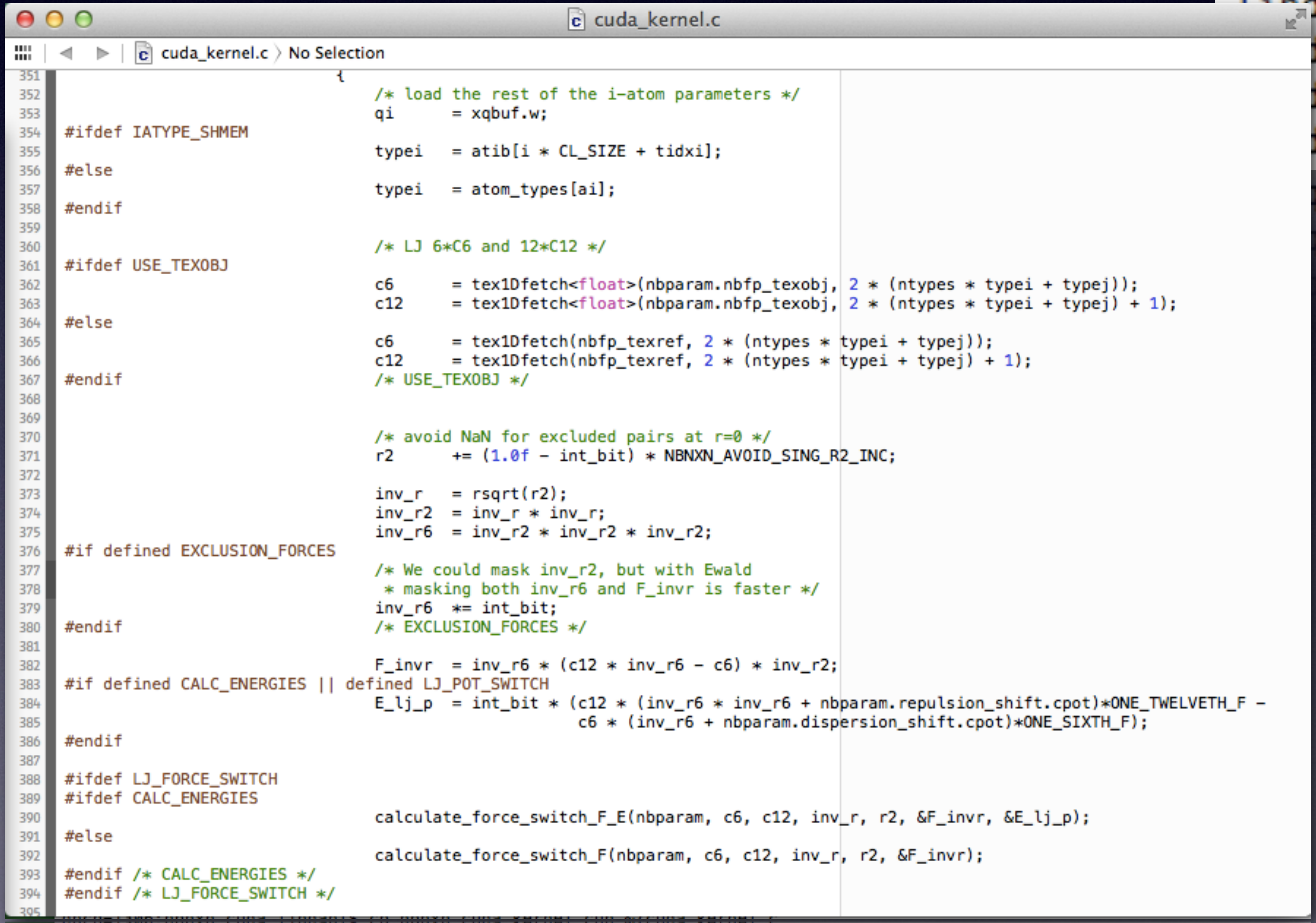
A
CPU
Pair Search
Bonded F
Non-bonded F
PME
Other Forces
Reduce Forces
Integration, Constraints
B
CPU
GPU
C
CPU
GPU
D
CPU
GPU
E
CPU
GPU

Pair-search &
domain-decomposition
every 10-250 steps
MPI comm: receive non-local x
MPI comm: send non-local F
MD step
CPU
GPU
DD comm
Domain decomp.
Pair search
3D FFT comm
constraint comm
PME mesh F
Other F
Wait non-local F
Wait local F
Integration Constraints
H2D pair list
H2D local x,q
H2D nonlocal x,q
A
Balance PME – non-bonded
D2H nonlocal F, E
D2H local F
Local stream
Non-local stream (high priority)
List pruning
preempted by non-local kernel
Local non-bonded F
Rolling prune
Clear buffers
Bonded F
Non-local non-bonded F
B
Balance DD + search with inner list pruning cost
Average CPU-GPU overlap: 70-90% per step

The big gain of heterogeneous acceleration: Very little CUDA required



```
n_cuda lindahl$ ls -ltar

 lindahl   staff   13012 Apr 18 15:10 nbnxn_cuda_types.h
 lindahl   staff    9155 Apr 18 15:10 nbnxn_cuda_kernels.cuh
 lindahl   staff   21576 Apr 18 15:10 nbnxn_cuda_kernel_utils.cuh
 lindahl   staff   20945 Apr 18 15:10 nbnxn_cuda_kernel.cuh
 lindahl   staff    1965 Apr 18 15:10 CMakeLists.txt
 lindahl   staff   39049 Apr 18 15:10 nbnxn_cuda_data_mgmt.cu
 lindahl   staff    3667 Apr 18 15:10 nbnxn_cuda.h
    dahl   staff   30920 May 22 09:13 nbnxn_cuda.cu
    dahl   staff    2686 May 22 09:13 ..
    dahl   staff     340 May 22 09:13 .
```



```
                           {
                                /* load the rest of the i-atom parameters */
                                qi      = xqbuf.w;
#ifdef IATYPE_SHMEM
                                typei   = atib[i * CL_SIZE + tidxi];
#else
                                typei   = atom_types[ai];
#endif

                                /* LJ 6*C6 and 12*C12 */
#ifdef USE_TEXOBJ
                                c6      = tex1Dfetch<float>(nbparam.nbfp_texobj, 2 * (ntypes * typei + typej));
                                c12     = tex1Dfetch<float>(nbparam.nbfp_texobj, 2 * (ntypes * typei + typej) + 1);
#else
                                c6      = tex1Dfetch(nbfp_texref, 2 * (ntypes * typei + typej));
                                c12     = tex1Dfetch(nbfp_texref, 2 * (ntypes * typei + typej) + 1);
#endif
                                /* USE_TEXOBJ */


                                /* avoid NaN for excluded pairs at r=0 */
                                r2      += (1.0f - int_bit) * NBNXN_AVOID_SING_R2_INC;

                                inv_r   = rsqrt(r2);
                                inv_r2  = inv_r * inv_r;
                                inv_r6  = inv_r2 * inv_r2 * inv_r2;
#if defined EXCLUSION_FORCES
                                /* We could mask inv_r2, but with Ewald
                                 * masking both inv_r6 and F_invr is faster */
                                inv_r6  *= int_bit;
#endif
                                /* EXCLUSION_FORCES */

                                F_invr  = inv_r6 * (c12 * inv_r6 - c6) * inv_r2;
#if defined CALC_ENERGIES || defined LJ_POT_SWITCH
                                E_lj_p  = int_bit * (c12 * (inv_r6 * inv_r6 + nbparam.repulsion_shift.cpot)*ONE_TWELVETH_F -
                                                     c6 * (inv_r6 + nbparam.dispersion_shift.cpot)*ONE_SIXTH_F);
#endif

#ifdef LJ_FORCE_SWITCH
#ifdef CALC_ENERGIES
                                calculate_force_switch_F_E(nbparam, c6, c12, inv_r, r2, &F_invr, &E_lj_p);
#else
                                calculate_force_switch_F(nbparam, c6, c12, inv_r, r2, &F_invr);
#endif /* CALC_ENERGIES */
#endif /* LJ_FORCE_SWITCH */
```

A total of ~3000 lines of CUDA, compared to ~2M lines of C++

... so we wrote OpenCL kernels too!

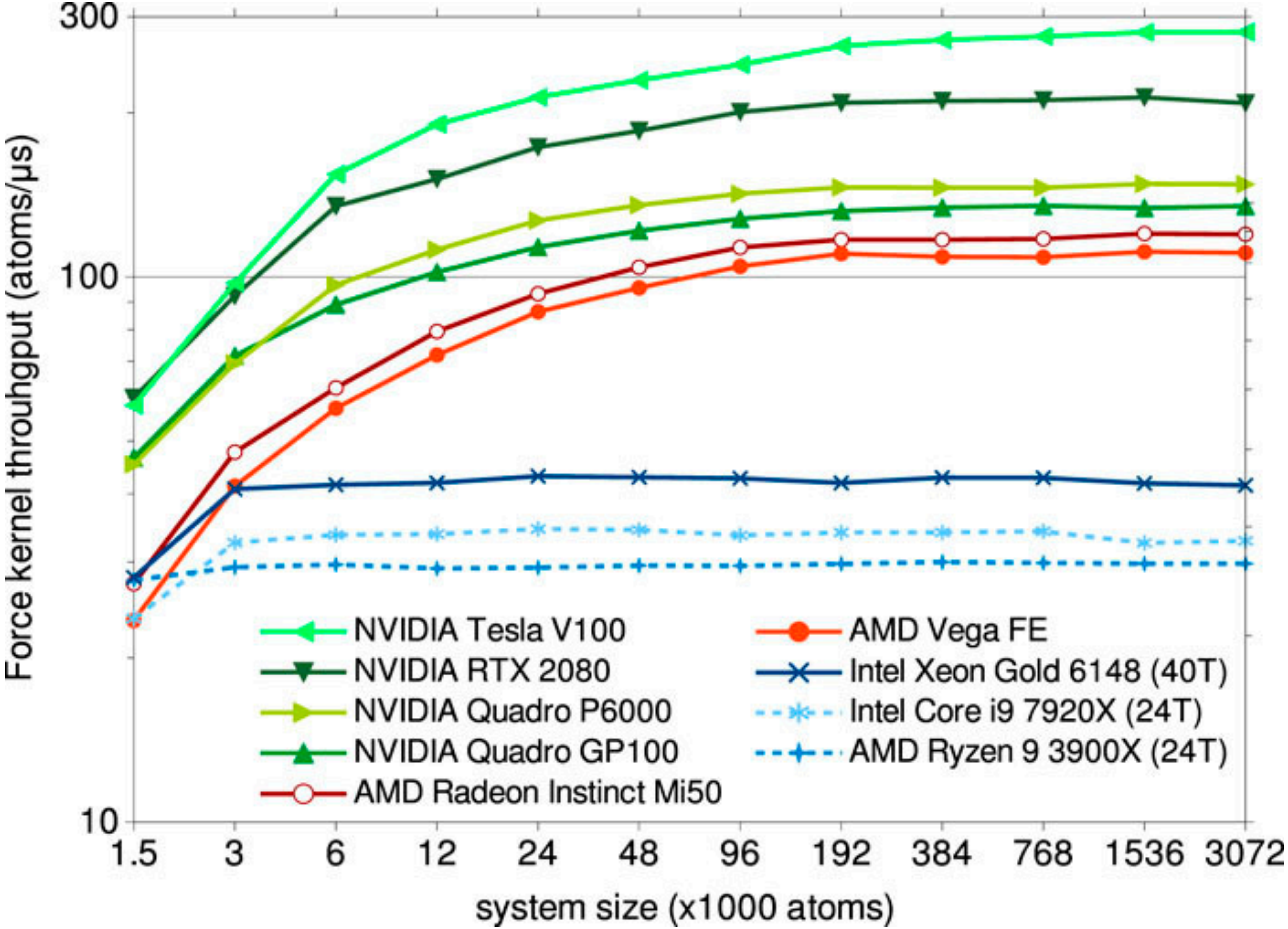
Make good use of both GPU & CPU

- Use the CPU to pre-calculate or optimize data structures, so there is less work for the GPU to do in your kernels
- Easier to implement more complex optimization on CPU
- Advanced multi-node domain decomposition easier on CPU
- Run some parts of the algorithm on the CPU (avoid wasting flops)

1. It's important to keep the GPU busy
2. … but it doesn't have to be busy 100% of the time!
3. A CUDA GPU running at 100% will get hot, and clock down
4. NVML "application clocks" effectively overclock the GPU on-the-fly when you have less than 100% utilization

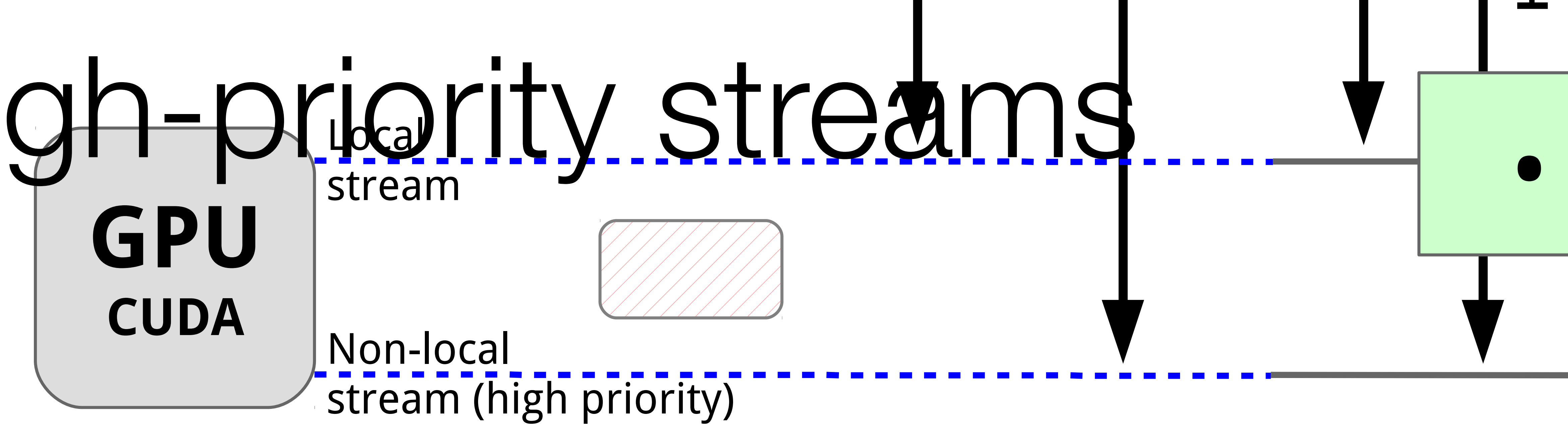
Think of a node as a collection of compute & communication devices - use them all!

Kernel latency is key for (heterogeneous) acceleration



A
CPU
Pair Search
Bonded F
Non-bonded F
PME
Other Forces
Reduce Forces
Integration, Constraints
B
CPU
GPU
C
CPU
GPU
D
CPU
GPU
E
CPU
GPU

Exploiting multiple & high-priority streams



Stream that only needs local data can start directly, but can be preempted by the high-priority nonlocal data kernel

When remote data is delivered, handle it immediately so it can be returned faster

Exploiting multiple & high-priority streams

Stream that only needs local data can start directly, but can be preempted by the high-priority nonlocal data kernel

MPI comm: receive non-local x
Pair-search & domain-decomposition every 10-250 steps
MPI comm: send non-local F
MD step
CPU
DD comm
Domain decomp.
Pair search
3D FFT comm
PME mesh F
Other F
Wait non-local F
Wait local F
constraint comm
Integration Constraints
H2D pair list
H2D local x,q
H2D nonlocal x,q
A
Balance PME – non-bonded
D2H nonlocal F, E
D2H local F
GPU
Local stream
List pruning
• • •
preempted by non-local kernel
Local non-bonded F
Rolling prune
Clear buffers
Non-local stream (high priority)
Bonded F
Non-local non-bonded F
B
Balance DD + search with inner list pruning cost
Average CPU-GPU overlap: 70-90% per step

When remote data is delivered, handle it immediately so it can be returned faster

Revisiting Amdahl's law - give GPU more work

The least parallel part of the code (or at least slowest piece of hardware) will eventually dominate execution completely and limit scaling

Thanks to heterogeneous parallelism and efficient CPU-side algorithms, GROMACS frequently outperforms GPU-only implementations - and yet we only need a few thousand lines of CUDA.

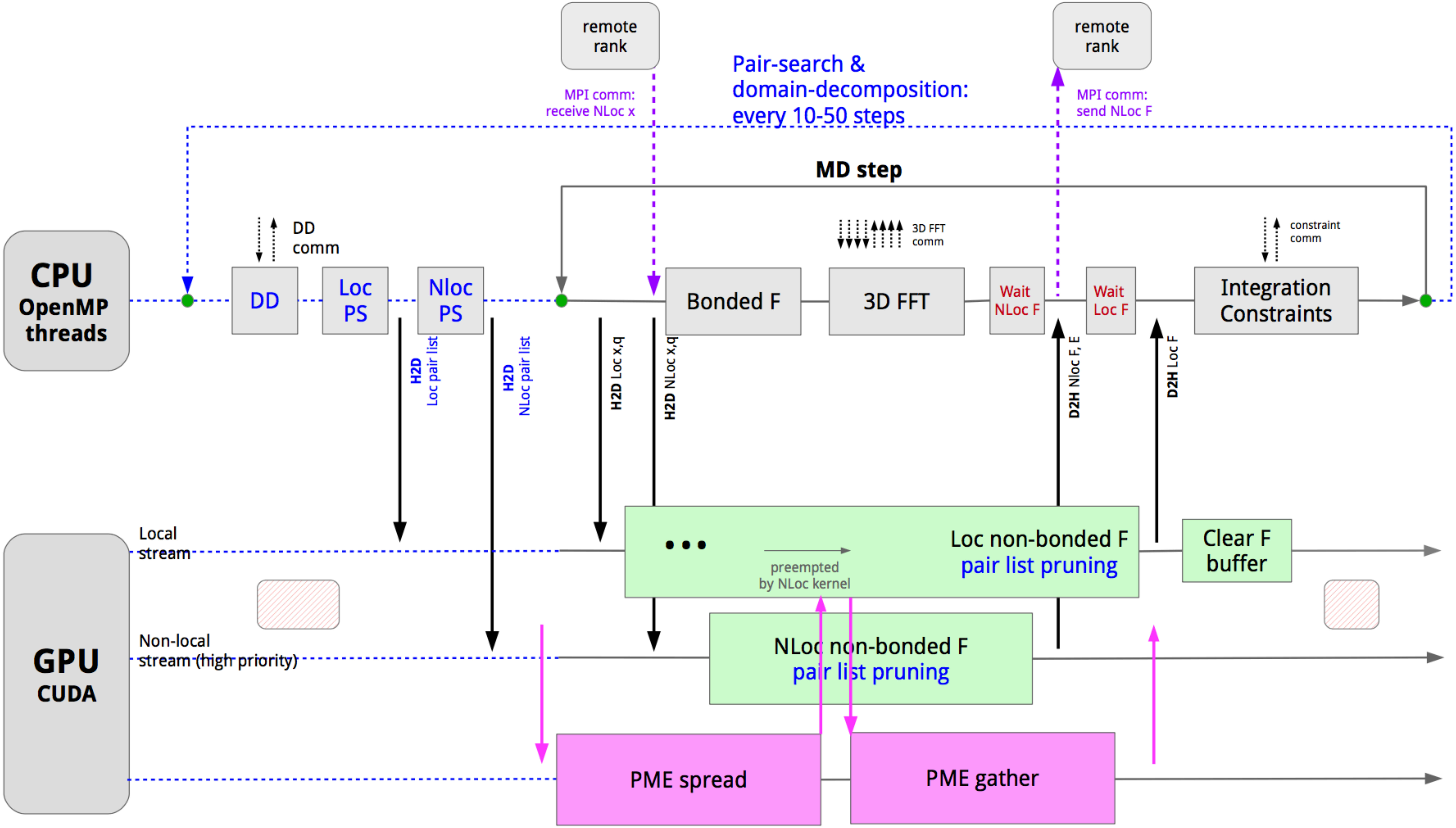
But… GPU performance grows faster than CPU performance, and sometimes we want to put a high-end GPU in an old low-end CPU box

Our CPUs used to wait for the GPUs, now it's often the opposite

A
CPU
Pair Search
Bonded F
Non-bonded F
PME
Other Forces
Reduce Forces
Integration, Constraints
B
CPU
GPU
C
CPU
GPU
D
CPU
GPU
E
CPU
GPU

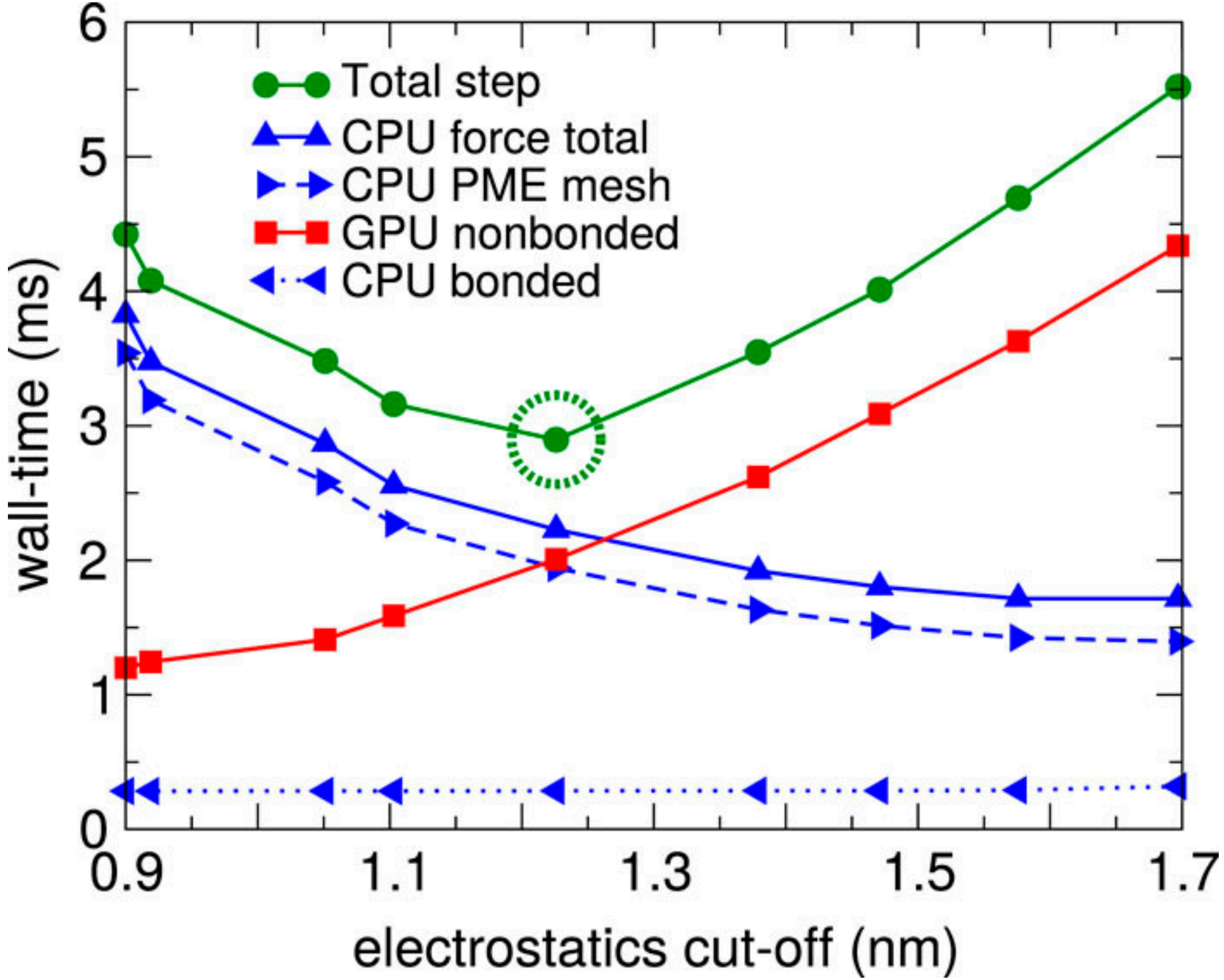
The new bottleneck (for slow CPUs) is the PME algorithm
3D grid spreading, FFTs, convolution, iFFT, interpolation

MPI comm: receive non-local x
Pair-search & domain-decomposition every 10-250 steps
MPI comm: send non-local F
MD step
DD comm
3D FFT comm
constraint comm
CPU
Domain decomp.
Pair search
PME mesh F
Other F
Wait non-local F
Wait local F
Integration Constraints
H2D pair list
H2D local x,q
H2D nonlocal x,q
A
Balance PME – non-bonded
D2H nonlocal F, E
D2H local F
GPU
Local stream
Non-local stream (high priority)
List pruning
• • •
preempted by non-local kernel
Local non-bonded F
Rolling prune
Clear buffers
Bonded F
Non-local non-bonded F
B
Balance DD + search with inner list pruning cost
Average CPU-GPU overlap: 70-90% per step



PME offload into separate stream

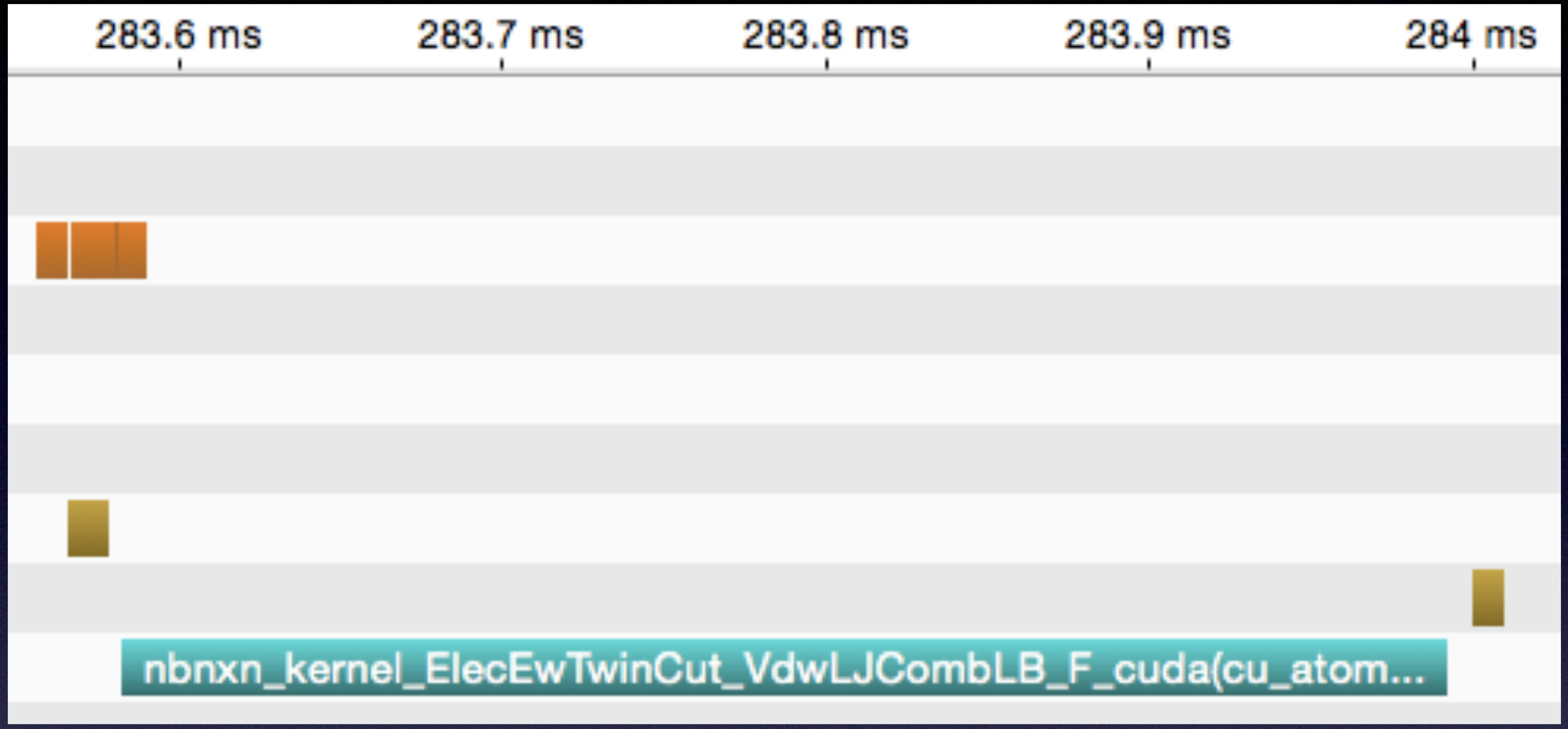
Load-balance the *algorithm* (not just work) between CPU & GPU



GPU Timestep profile before/after PME offload (P100)

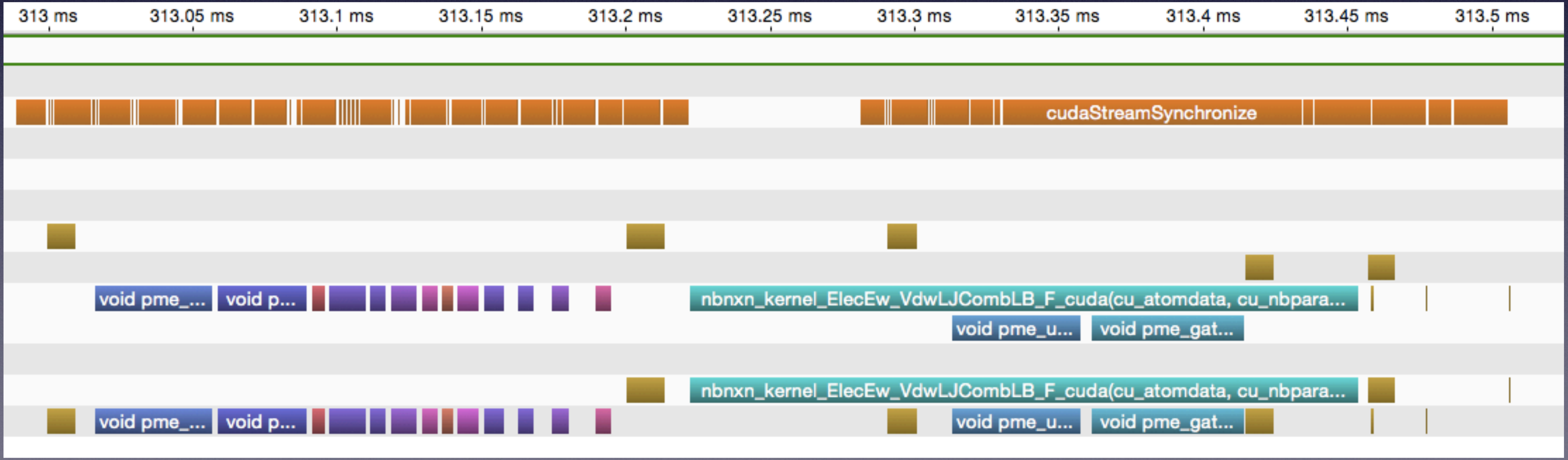
Highly challenging small system (25k atoms), very fast iterations.

Much harder, but important for multi-GPU scaling

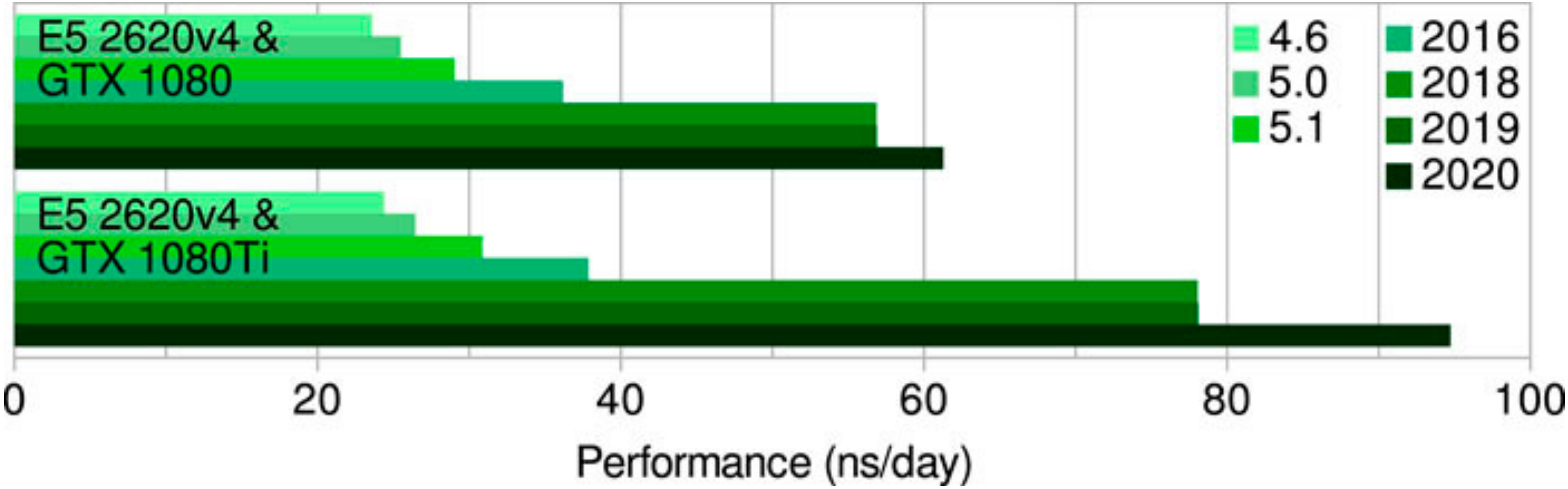


450μs for a complete step - note x scale!

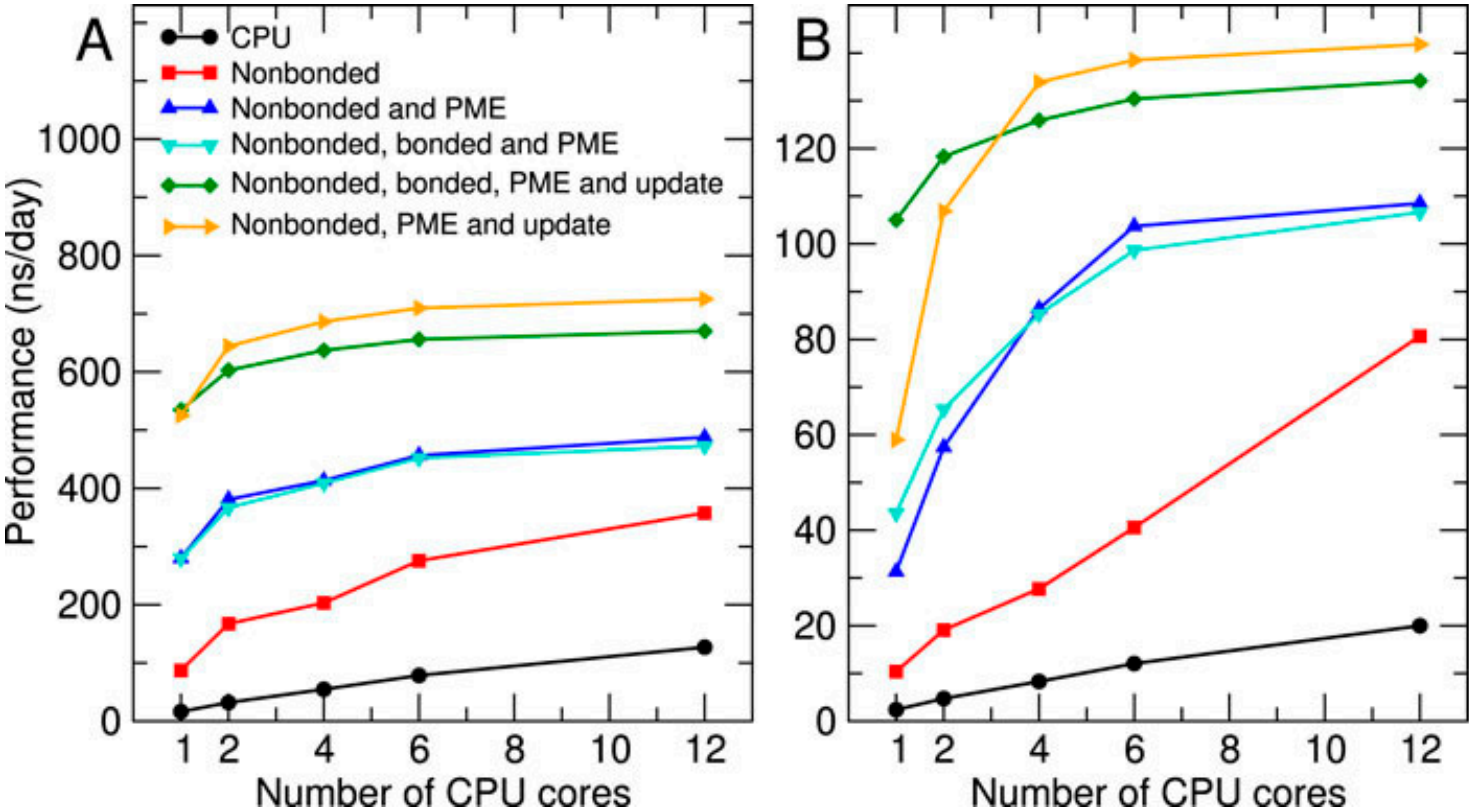
Harder to retain full compute throughput than communication BW when scaling to small sys. Scheduling limited below - kernels should overlap. Lauch operations limiting us.



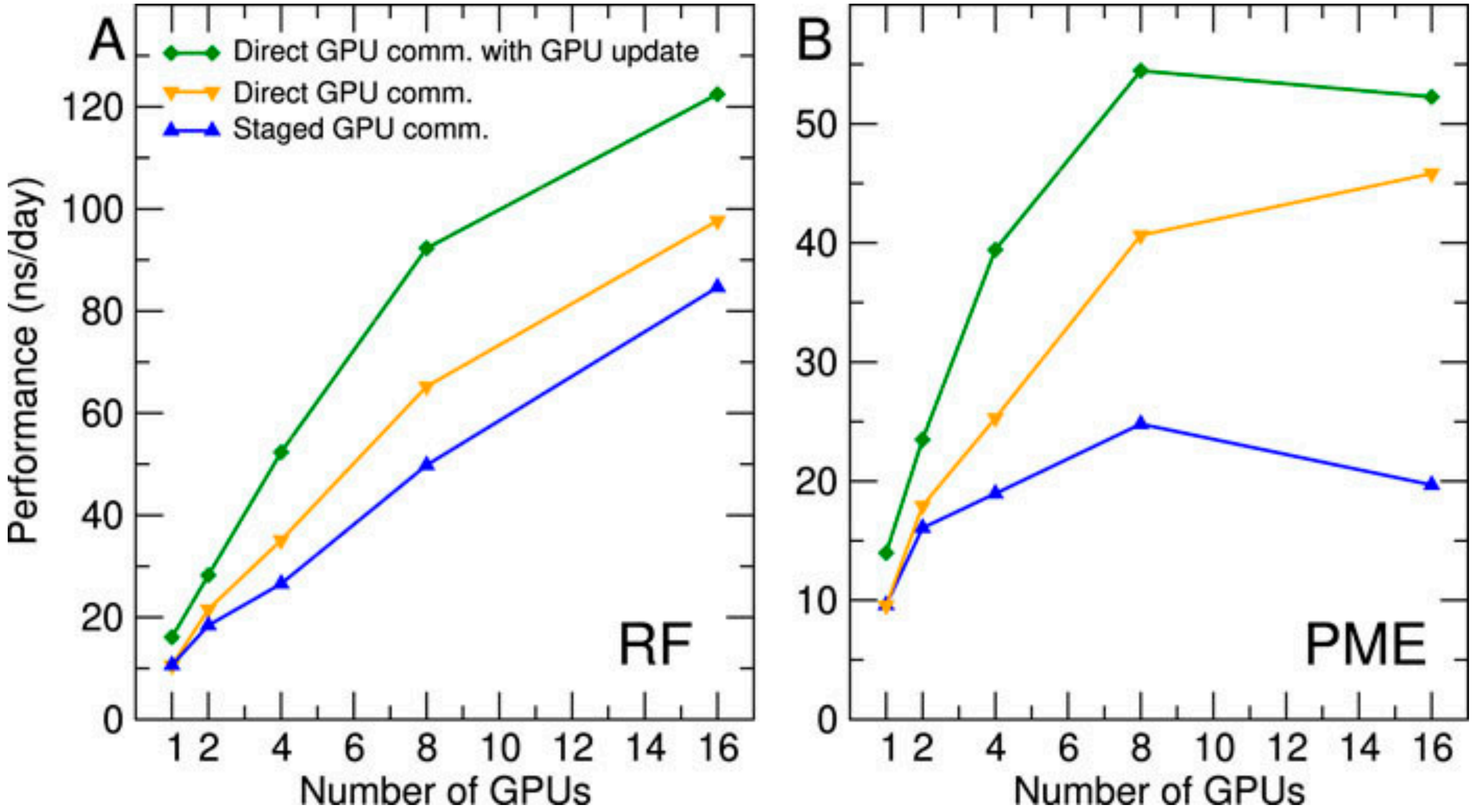
Even codes that have been tuned for ~20 years on CPUs and ~10 years on GPUs can get great performance gains just from better algorithms & implementations



We solved our issue of fast-CPU-dependency: Fast with a single core per GPU, even faster with many



Heterogeneous parallelisation provides good scaling even for the highly latency-sensitive algorithms in molecular dynamics

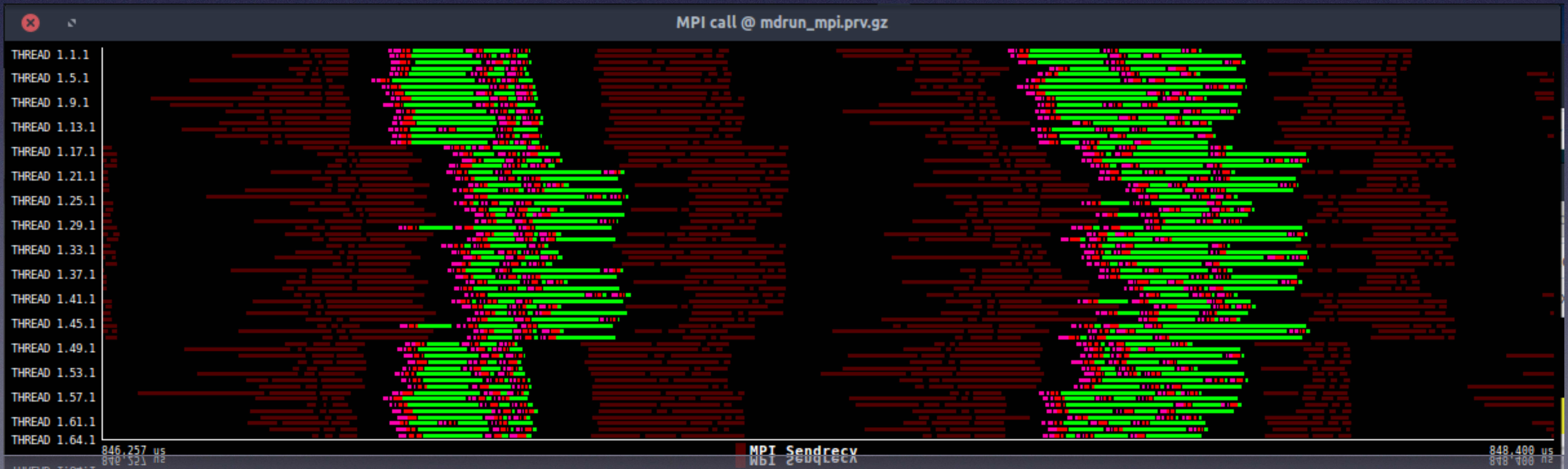


Strong caling issues - challenges at 100μs per iteration

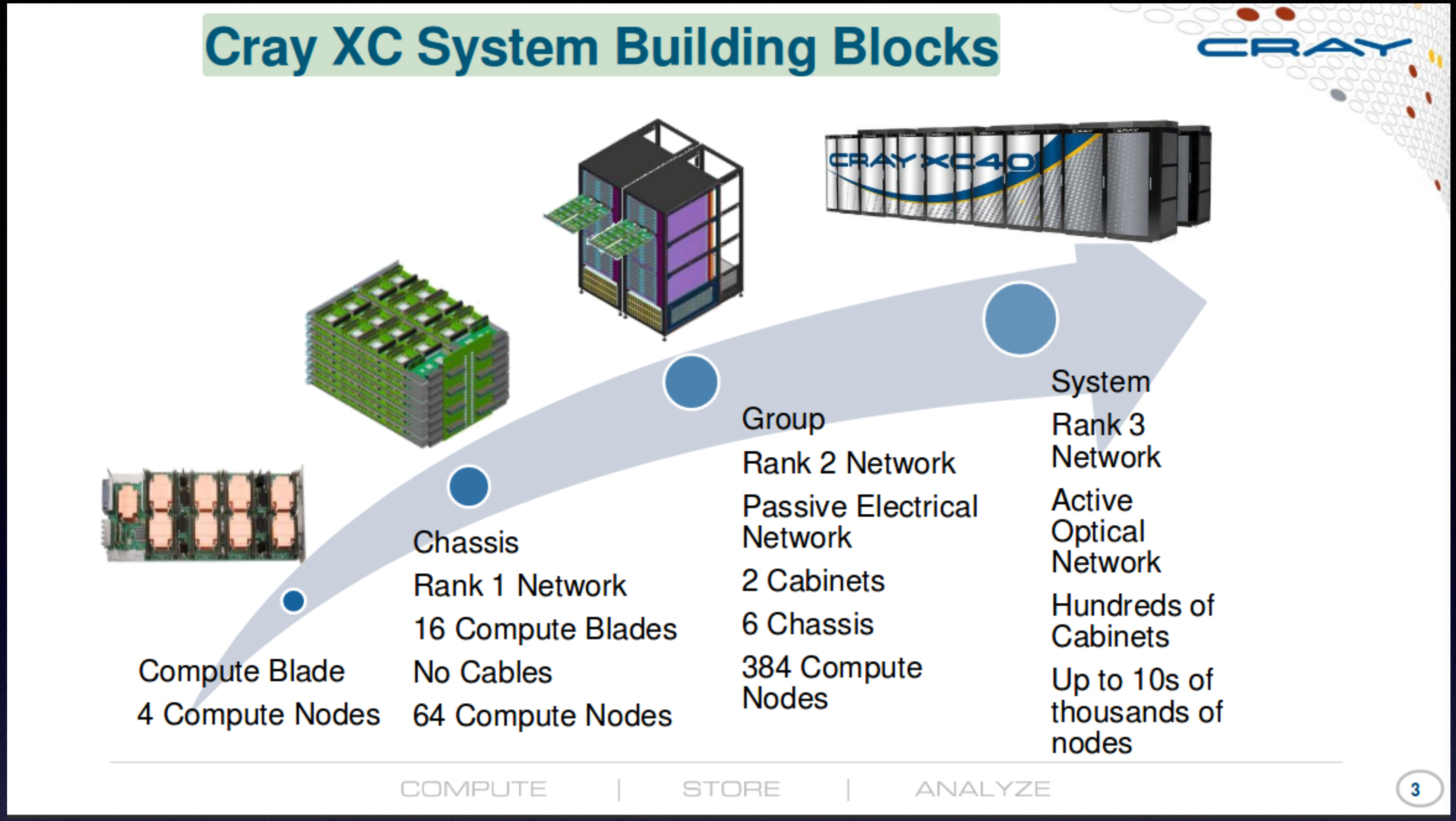
- The 3D-FFT in PME
- MPI overhead - we need MPI_Put_notify()
- OpenMP barriers take significant time
- Load imbalance
- CUDA API overhead can be 50% of CPU time
- Too many GROMACS options to tweak manually

Large performance loss due to imbalance and network speed variation on Cray XC (interference from other jobs on the “smart” network)

<1 millisecond



It would help a lot to have more control over node placement



Cray XC System Building Blocks

Compute Blade
4 Compute Nodes

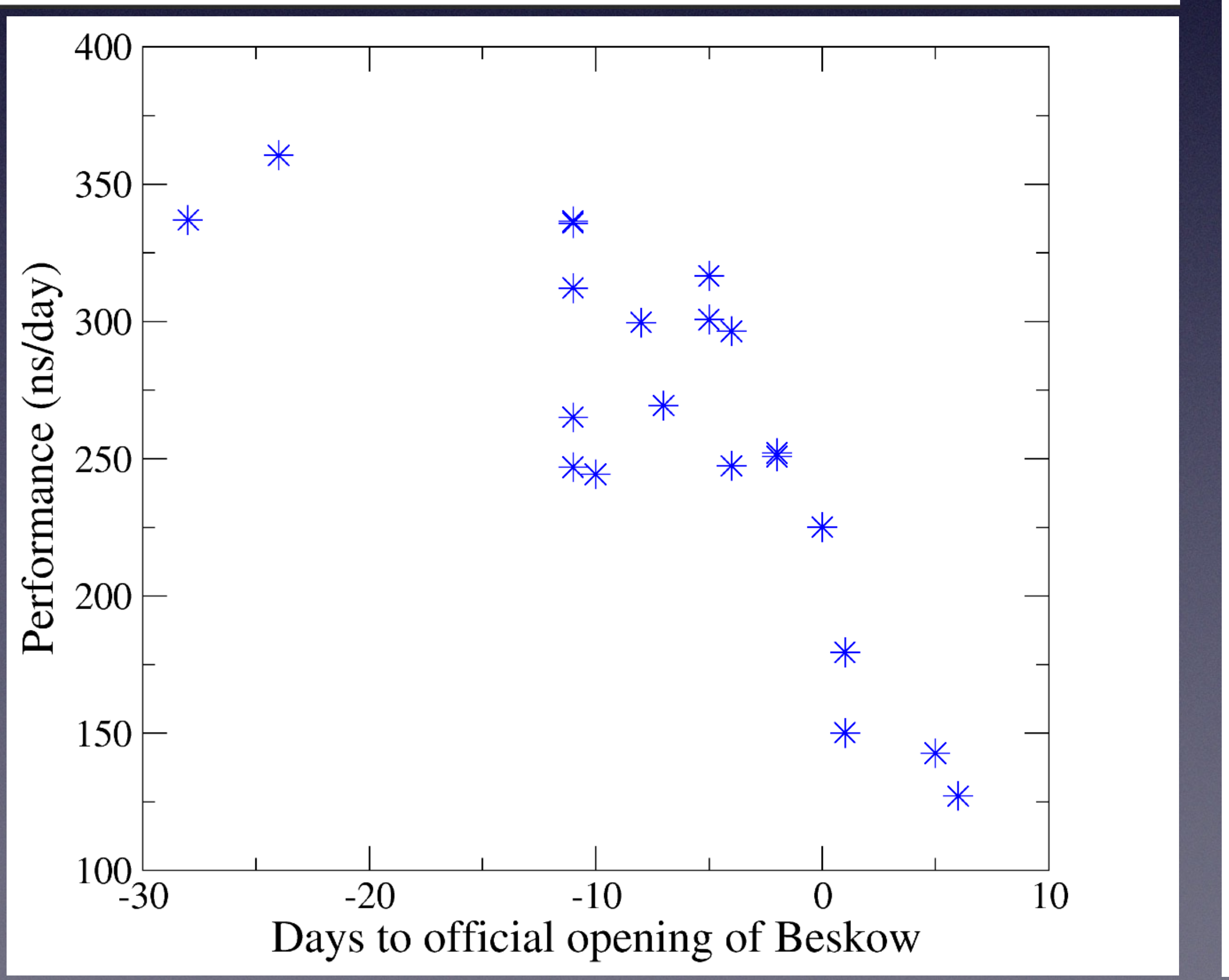
Chassis
Rank 1 Network
16 Compute Blades
No Cables
64 Compute Nodes

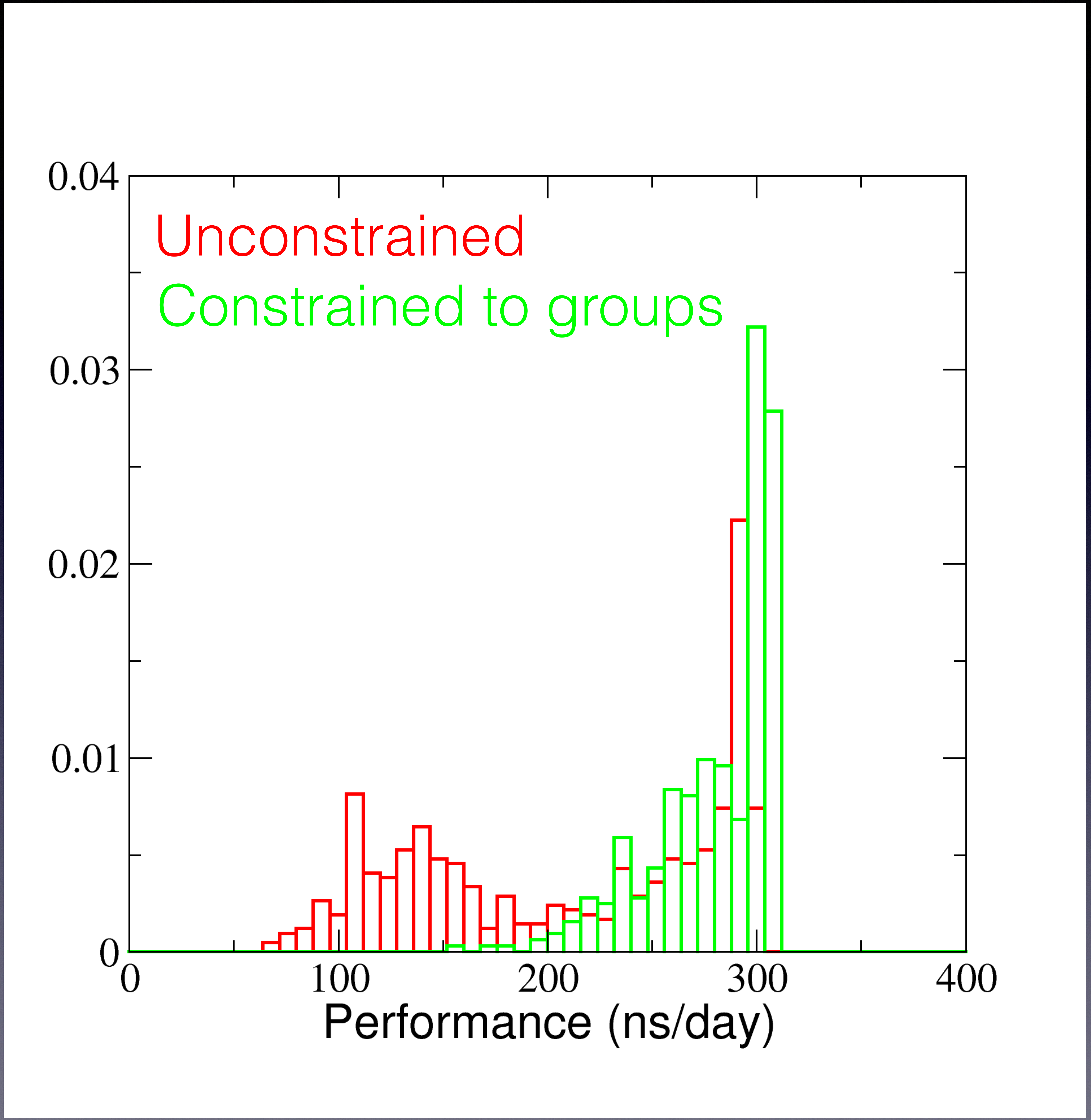
Group
Rank 2 Network
Passive Electrical Network
2 Cabinets
6 Chassis
384 Compute Nodes

System
Rank 3 Network
Active Optical Network
Hundreds of Cabinets
Up to 10s of thousands of nodes

COMPUTE | STORE | ANALYZE

3





Intra-rank parallelisation: OpenMP today, future ?

Efficient current parallelization of all algorithms using MPI + OpenMP

OpenMP is (performance) portable, but limited:

- No way to run parallel tasks next to each other
- No binding of threads to cores (cache locality)

Need for a better threading model, requirements:

- Extremely low overhead barriers (all-all, all-1, 1-all)
- Binding of threads to cores
- Portable

We are convinced we are moving to a world where latency- and throughput-optimized units converge into the same chip - the future is heterogeneous!

Urgent need for better, standardized and portable HPC-focused task parallelism frameworks. We are looking into both ArgoBots and home-grown solutions.

Spend time with your algorithms, not just code tuning.

A single Skylake-EP node has 4096-fold parallelism.
Your code likely doesn't.

Think accelerators - because a modern CPU looks like an accelerator,
and they will likely converge to multiple units on one die in the future.

Heterogeneous parallelism uses all resources and provides architecture portability.

Fast-iteration codes are very sensitive to node placement,
and they need task parallelism sooner rather than later.

Fast-iteration coding for CUDA/AVX512/OpenCL/SYCL isn't hard - but new algorithms are.

You can accomplish miracles with more codes than you think,
but it takes 6-12 months - not an afternoon.

Theory & Computation is the new experiment!

Acknowledgments

GROMACS: Berk Hess, Szilard Pall, Paul Bauer, Andrey Alekseenko, Mark Abraham, Aleksei Iiupinov, John Eblen, Christian Wennberg, Joe Jordan, Viveca Lindahl, Magnus Lundborg, Sebastian Wingbermühle

NVIDIA: Alan Gray, Jon Lefman, Mark Berger, Duncan Poole, Jiri Kraus, Nikolay Markovskiy

INTEL: Roland Shultz, Heinrich Bockhorst

CSCS: Thomas Schulthess, Victor Holanda, Prashant Kanduri

PDC: Erwin Laure

Stiftelsen för Strategisk Forskning

erc European Research Council

Sjunde Ramprogrammet

National Institutes of Health

PRACE

SeRC Swedish e-Science Research Centre

Vetenskapsrådet

bioexcel Center of Excellence for Computational Biomolecular Research