Heterogeneous task scheduling of molecular dynamics in GROMACS

Szilárd Páll
pszilard@kth.se

HiHAT seminar
January 19, 2021

Acknowledgments

GROMACS team

Berk Hess

Artem Zhmurov

Erik Lindahl

Paul Bauer

Mark Abraham

Magnus Lundborg

Roland Schulz

Aleksei Yupinov

Alan Gray (NVIDIA)

Gaurav Garg (NVIDIA)

HW / code contrib

Funding

TOC

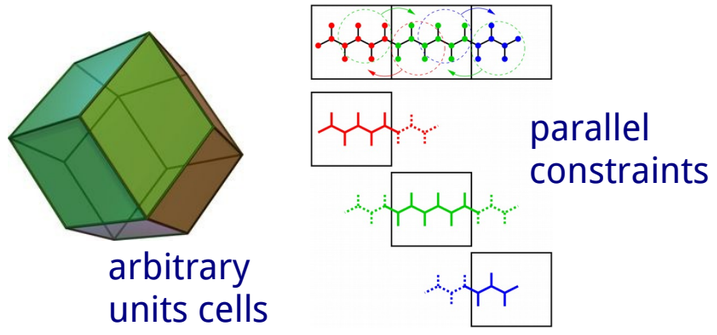
- Introduction
- Parallelizing bio-MD
- Heterogeneous tasking in GROMACS
- Challenges

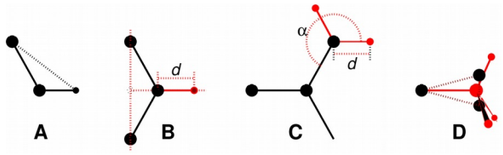
- Alan Gray:

  CUDA Graphs in GROMACS

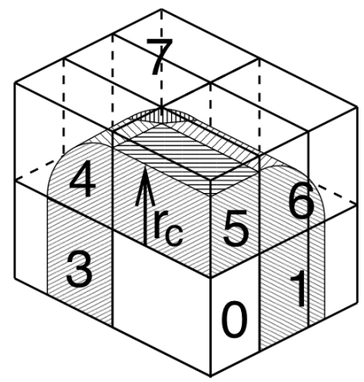
GROMACS
FAST. FLEXIBLE. FREE.

- **Classical MD** code
  - supports all major force-fields
  - broad algorithm support
- **Development**:
  - Stockholm Sweden
  - academic partners & vendor co-design partners ww
- **Large user base**:
  - 10k's academic & industry
  - deployed on most HPC resources
- **Open source**: LGPLv2
- **Open development:**
  - code review & bug-tracker:https://gitlab.com/gromacs

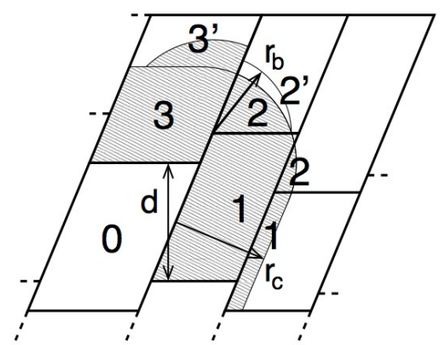




virtual interaction sites



Eighth shell
domain
decomposition



Triclinic unit cell with
load balancing and
staggered cell boundaries

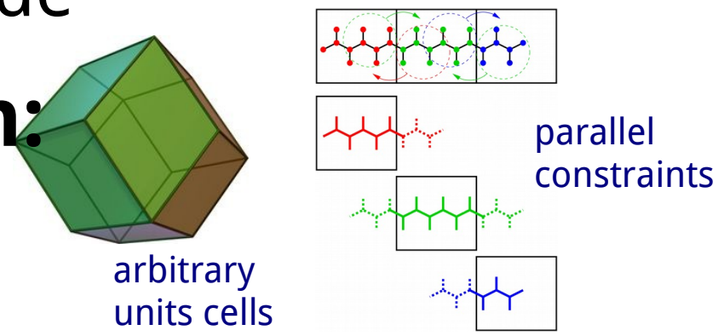
GROMACS

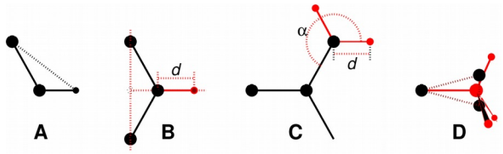
FAST. FLEXIBLE. FREE.

- **Focus on high performance:**

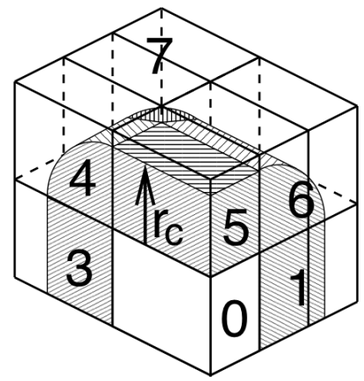
  efficient algorithms & highly-tuned parallel code

- **Bottom-up performance oriented design:**
  - absolute performance over “just scaling”
- **Heterogeneous parallelization by design**
  - for feature support/extensibility & performance
- **Portability**
  - broad CI testing, Linux distro integration
  - regular testing on all HPC arch
- **Code-base:** C++17, >1M LOC

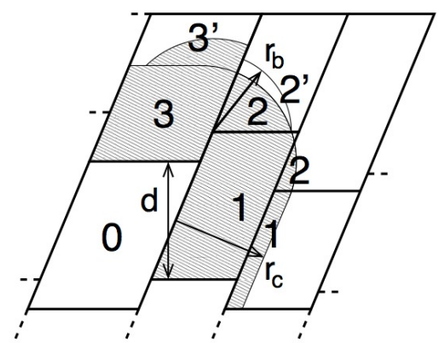




virtual interaction sites

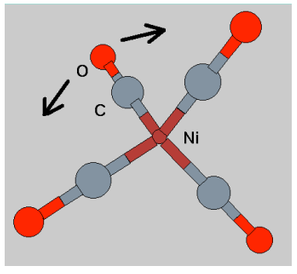


Eighth shell
domain
decomposition

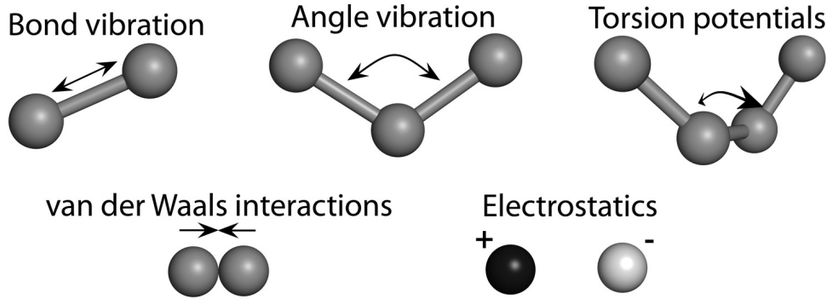


Triclinic unit cell with
load balancing and
staggered cell boundaries

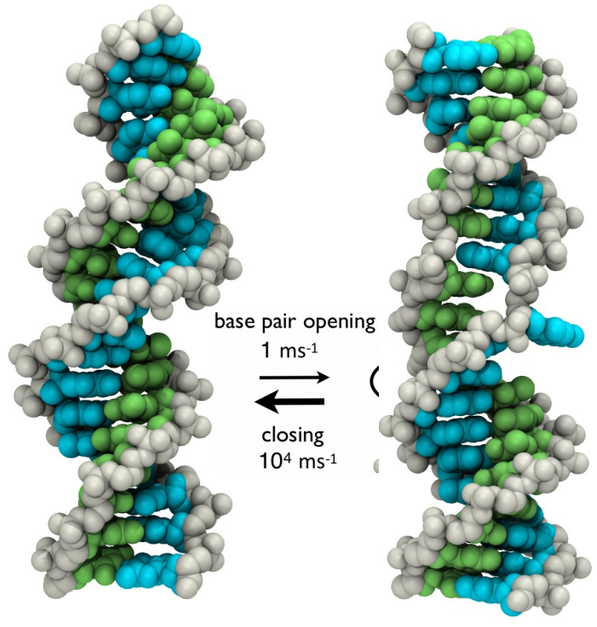
MD Timescale challenge



Physics



Chemistry

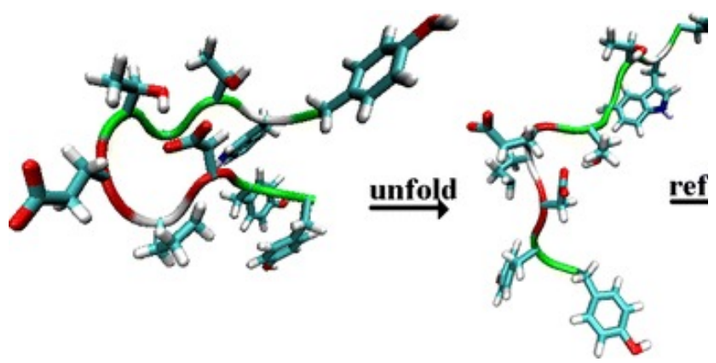


Biology

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

Simulations:

- high spatial/temporal detail
- sampling bottleneck
- model quality?



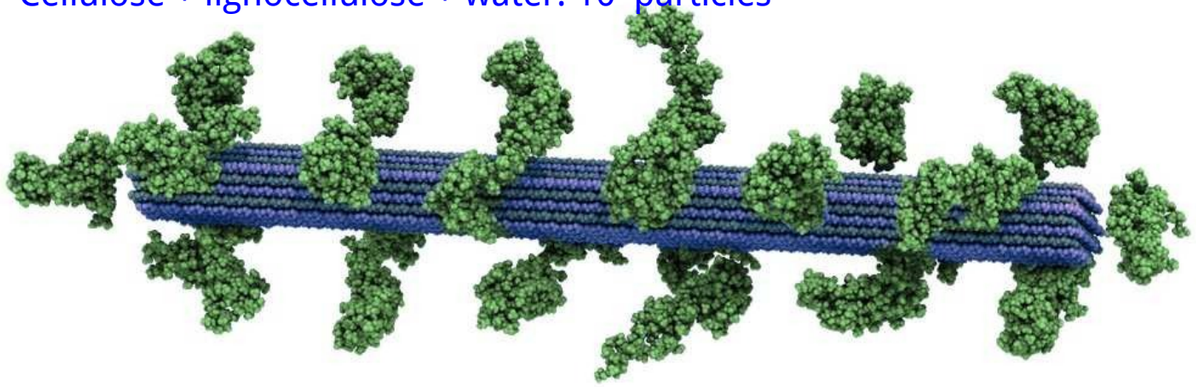
Laboratory experiments:

- lower detail
- higher efficiency
- high degree of averaging

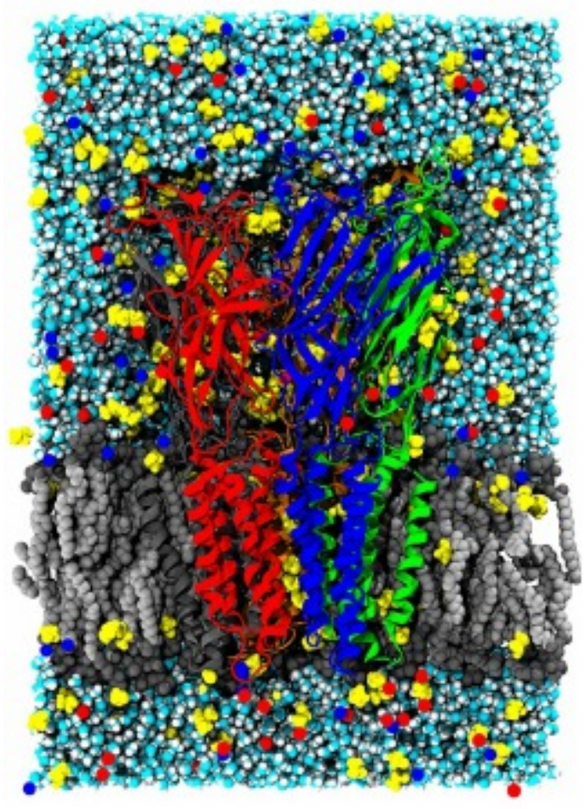
Molecular simulation: use-cases

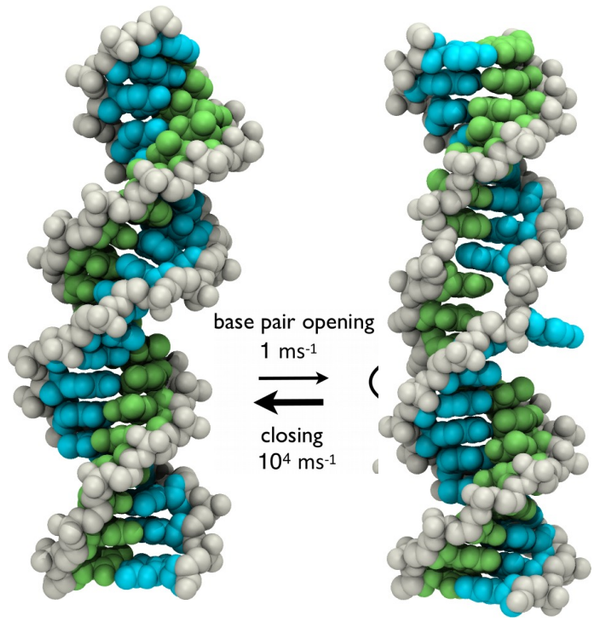
Biomolecular MD

Cellulose + lignocellulose + water: 10^7 particles



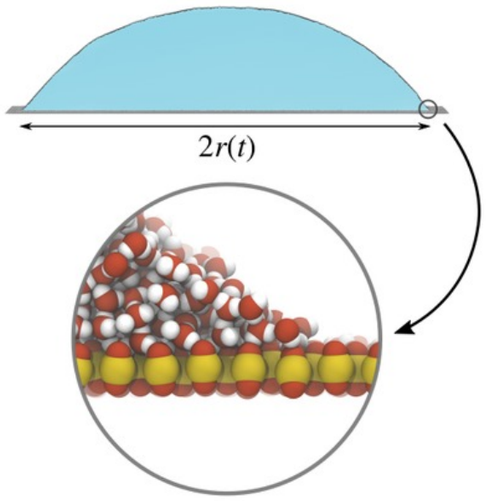
Membrane protein: 10^5 particles



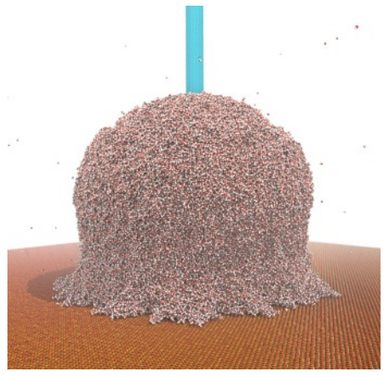


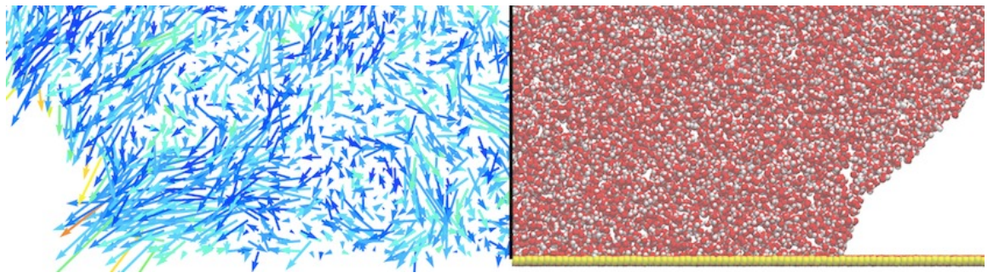
DNA base-pair opening: 10^4 particles

Materials MD

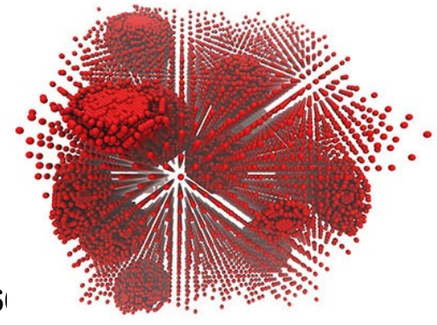


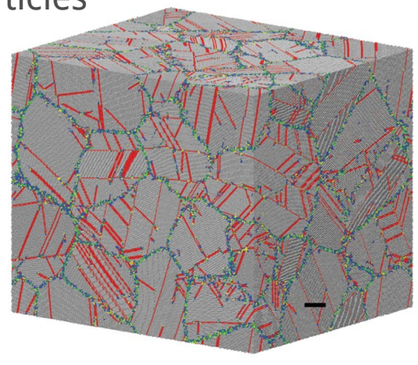
Contact line friction &
wetting dynamics
10^7 -10^9 particles





Nucleation in nano-crystals:
10^{10} -10^{12} particles





Molecular simulation: use-cases

Biomolecular MD

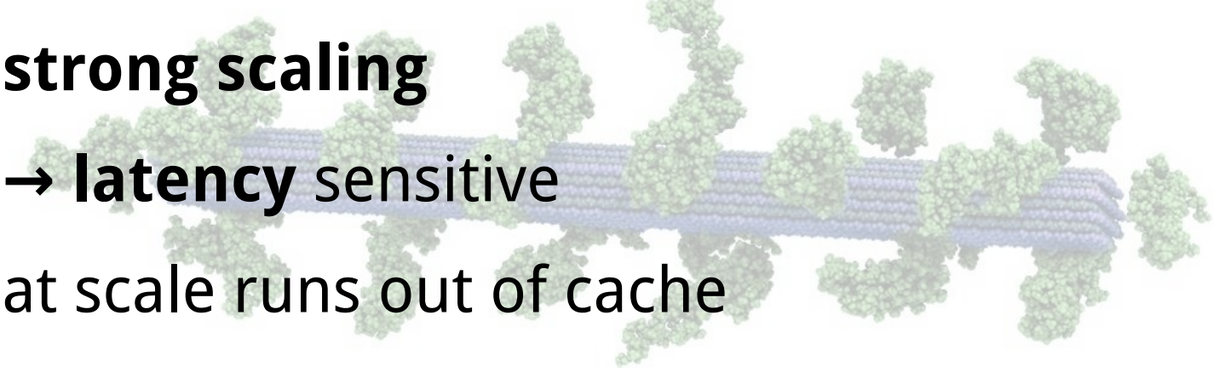
time-scale challenge

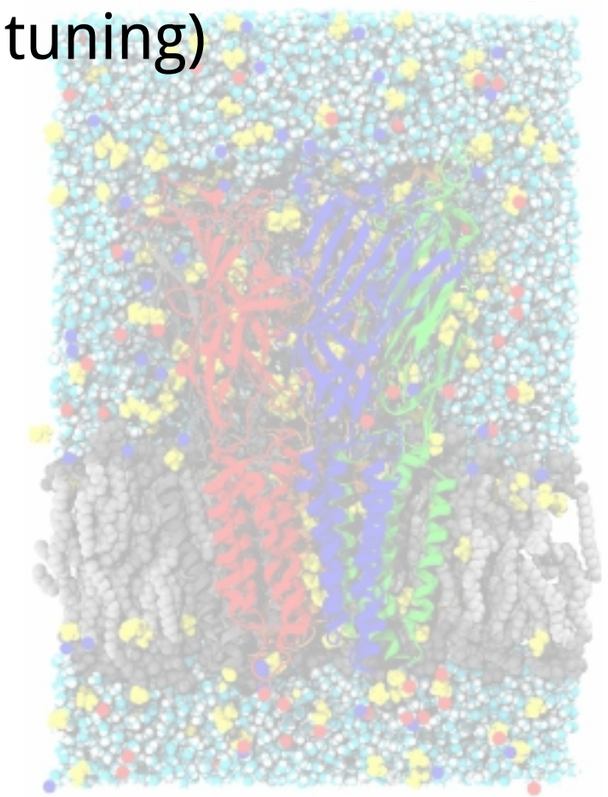
strong scaling

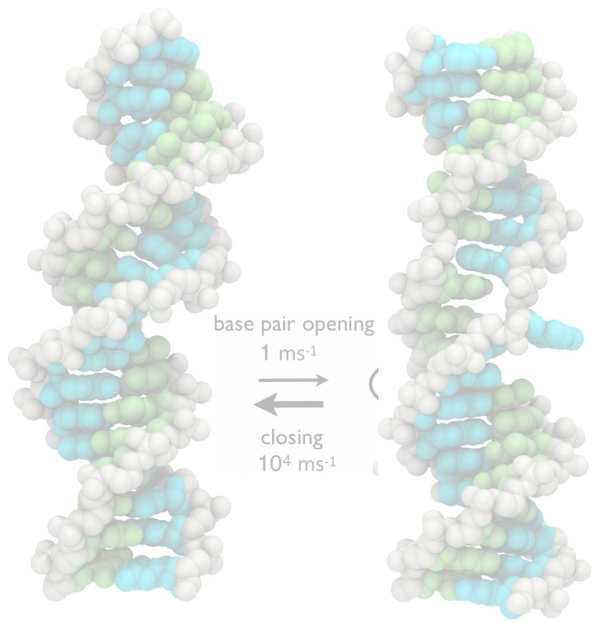
→ **latency** sensitive

at scale runs out of cache

→ strong benefit from high algorithm arithmetic intensity (SIMD, instruction tuning)







DNA base-pair opening: 10^4 particles

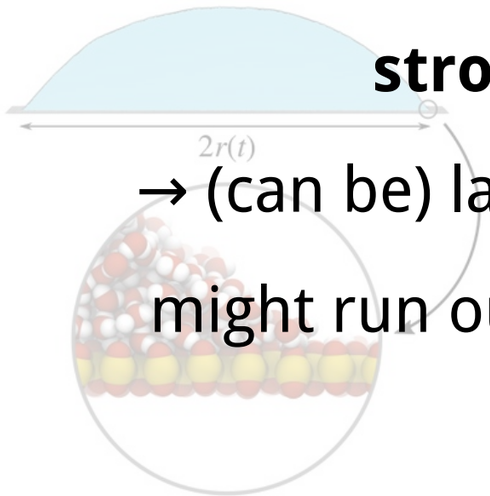
Materials MD

time- & length-scale challenge

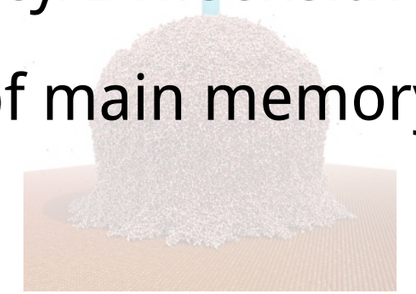
strong / weak scaling

→ (can be) latency/BW sensitive

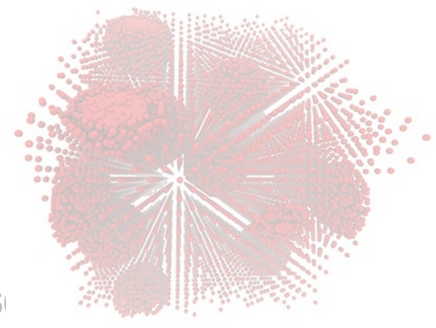
might run out of main memory

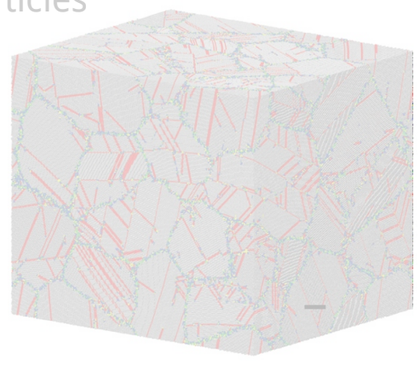


Contact line friction & wetting dynamics
10^8-10^9 particles

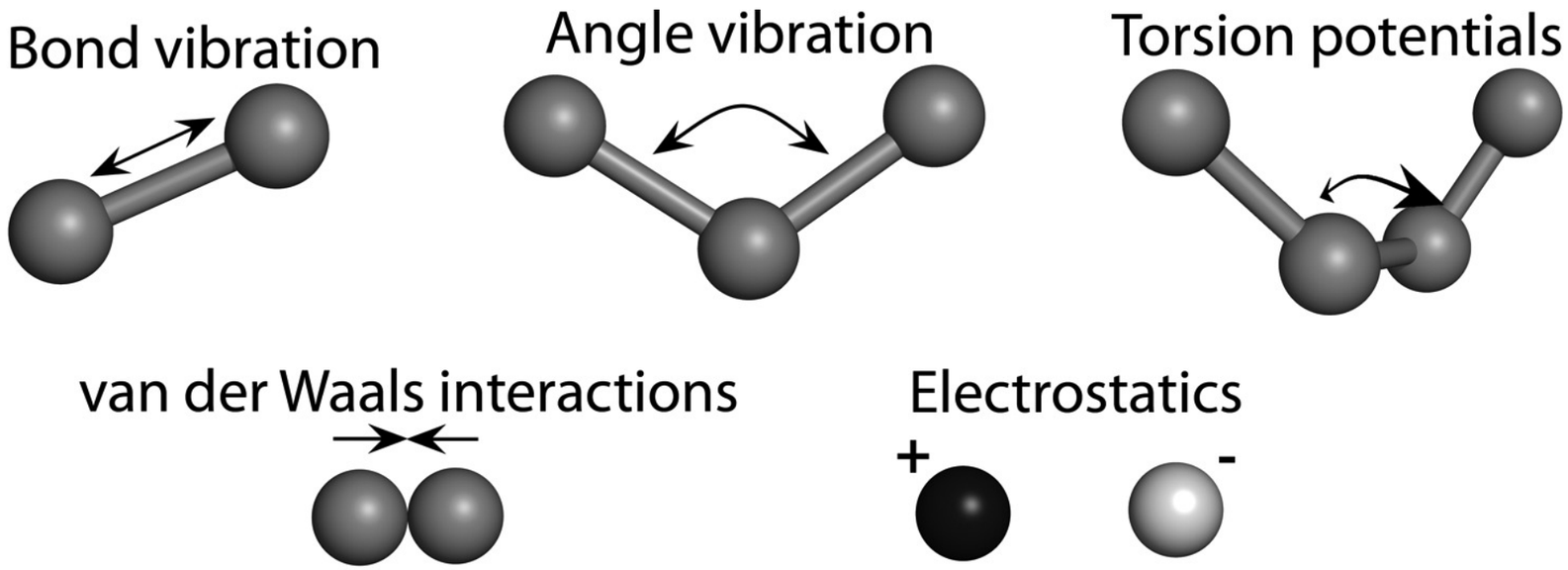


Nucleation in nano-crystals:
10^{10}-10^{12} particles





Main computate cost: calculating forces



$$U(\mathbf{r}) = \sum_{bonds} U_{bond}(\mathbf{r}) + \sum_{angles} U_{angle}(\mathbf{r}) + \sum_{dihs} U_{dih}(\mathbf{r})$$ Bonded

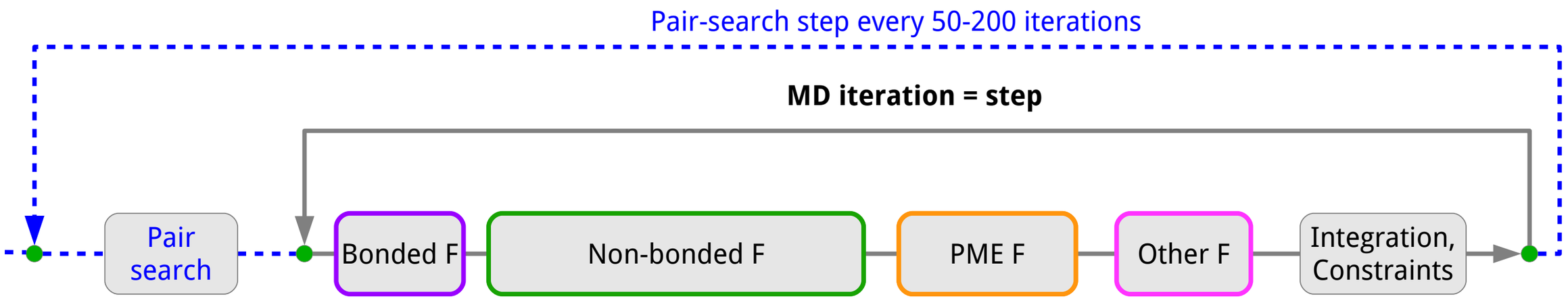
$$+ \sum_{i} \sum_{j>i} \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}} + \frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}}$$ Non-bonded

Over all atom-pairs!

Compute force on each particle

$$\mathrm{F}_i = -\frac{dU}{d\mathrm{r}_\mathrm{i}}$$

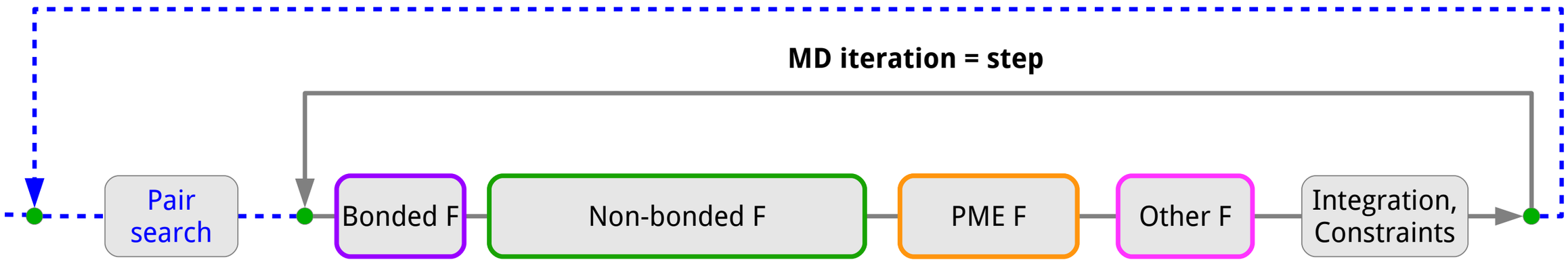
Molecular dynamics step

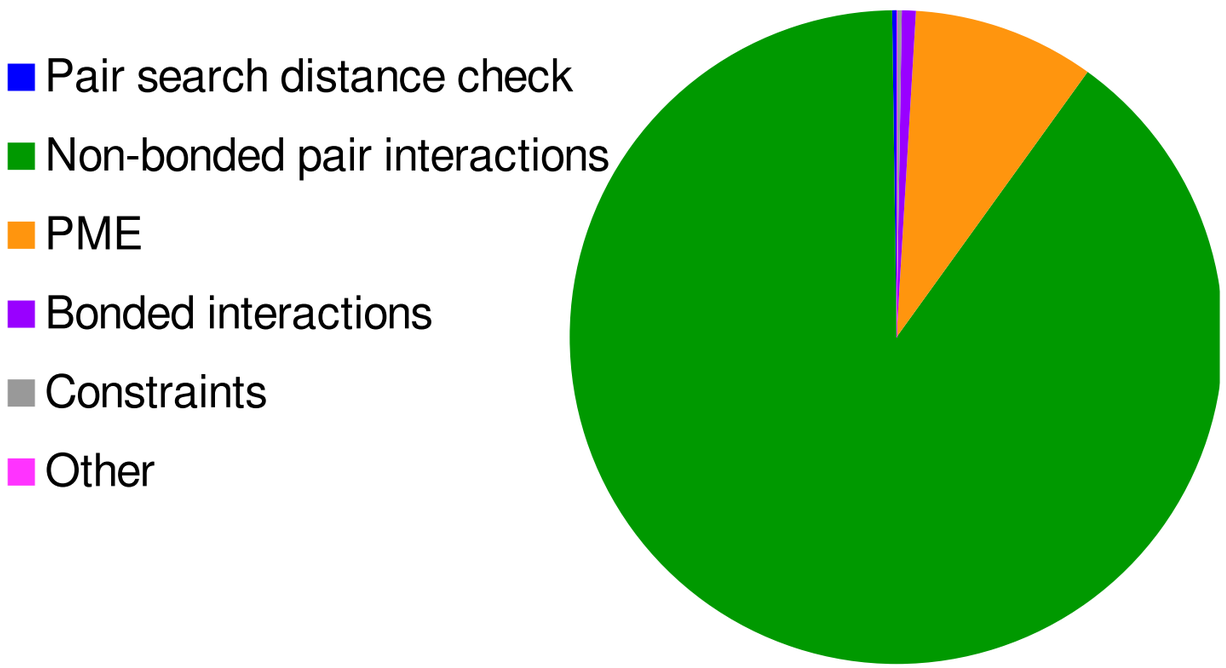


~ millisecond or less

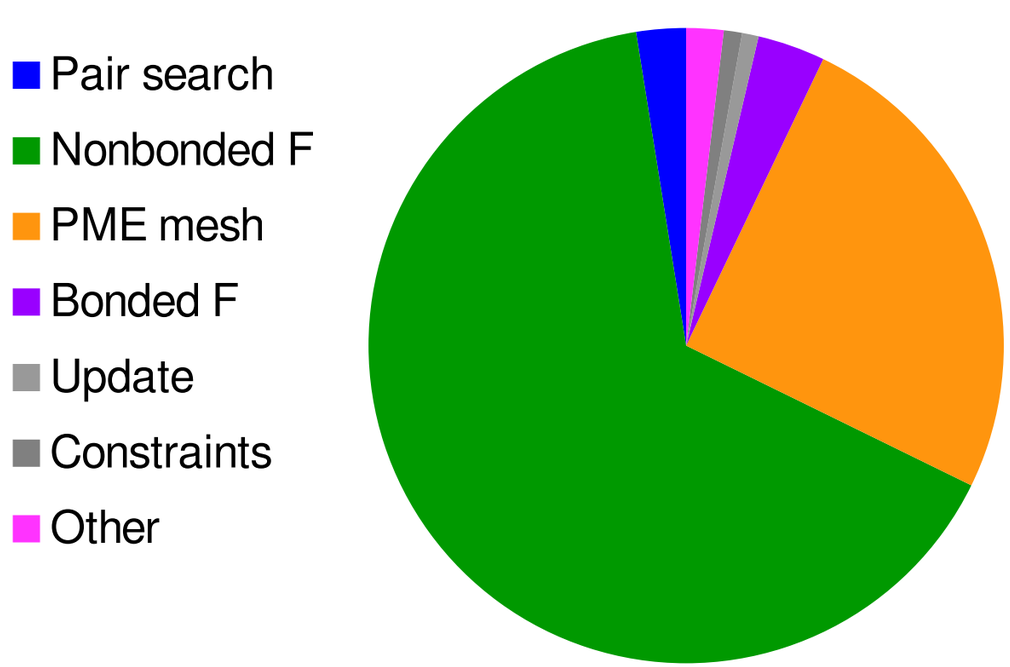
Goal: do it as fast as possible!

Computational costs



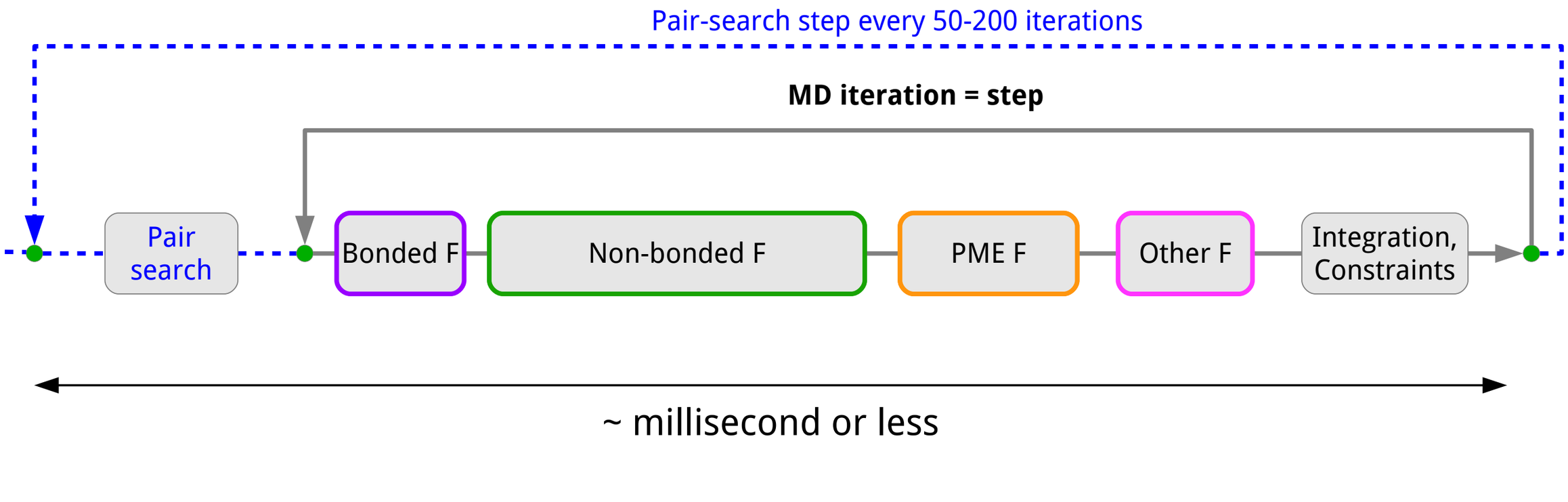


FLOPs in a typical simulation



Wall-time breakdown

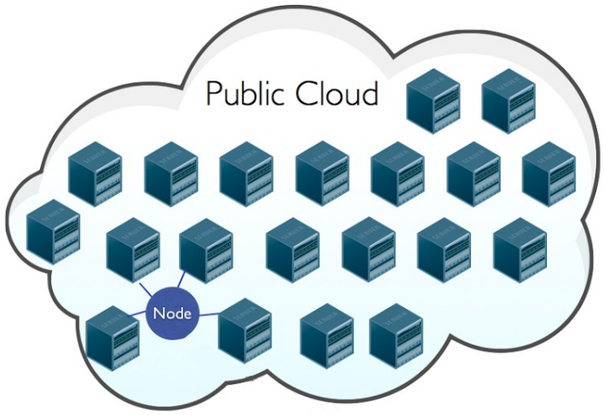
MD: strong scaling challenge



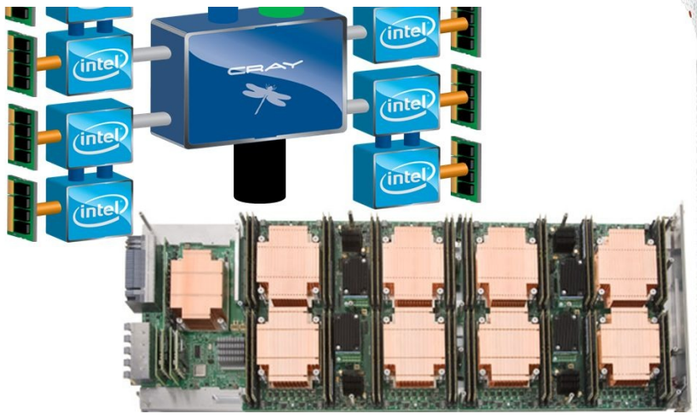
- Simluation vs real-world **time-scale gap**
  - Every simulation: 10^8 –10^{15} steps
  - Every step: 10^6 – 10^9 FLOPs

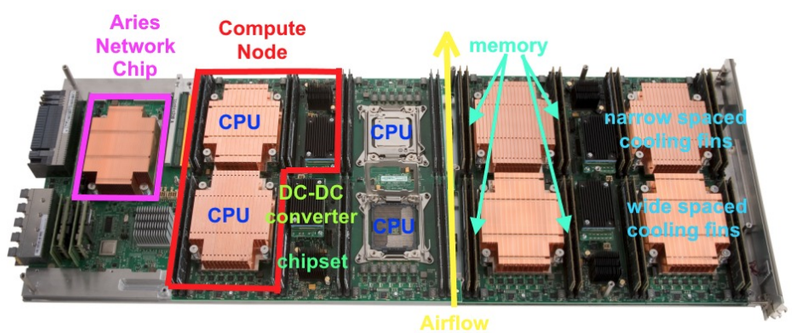
- MD codes at peak: ~**100 µs / step**
  - <100 atoms/core at peak
  - <10000 atoms / GPU

Multiple levels of hardware parallelism



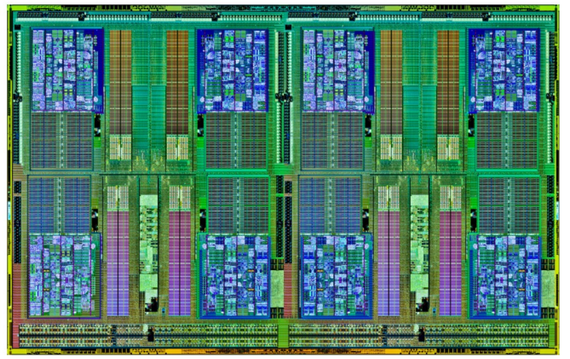
Compute cluster or cloud
Networked computers:
topology, bandwidth, latency



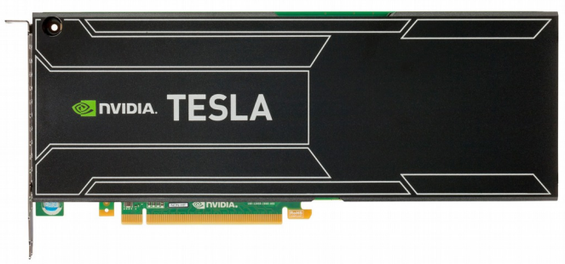


Compute node / workstation

NUMA topology, PCIe

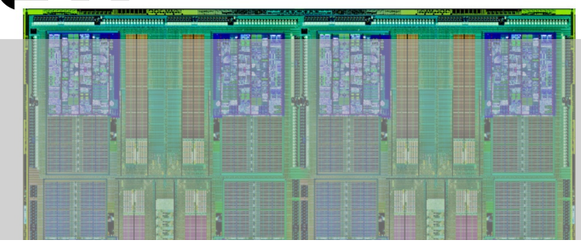


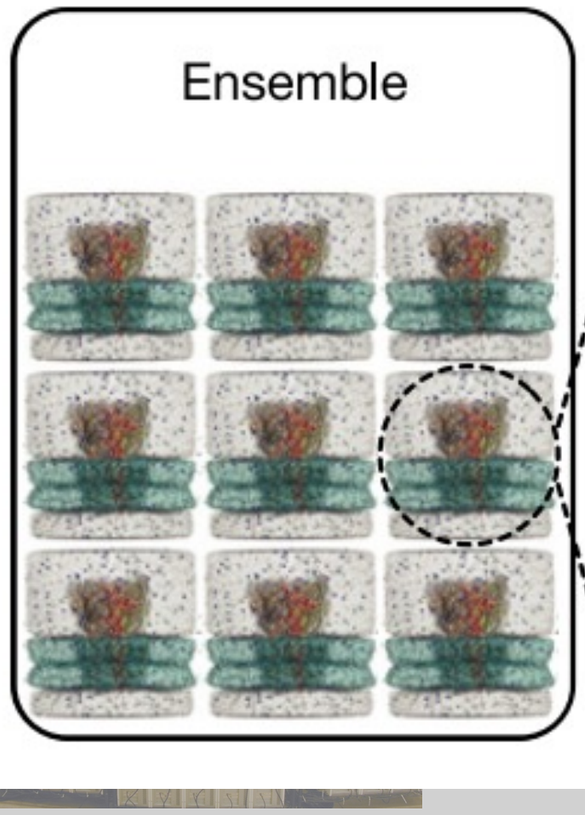
up to 512-bit vector units/core
=>
up to 16 single precison ops/clock

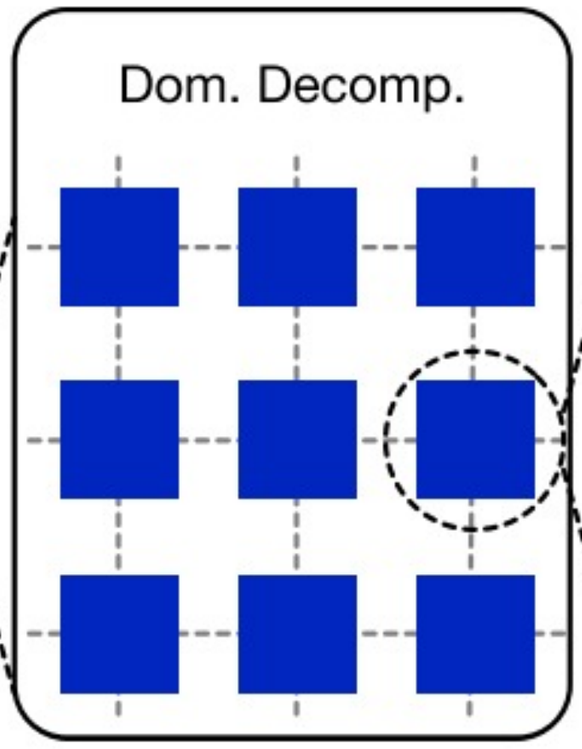


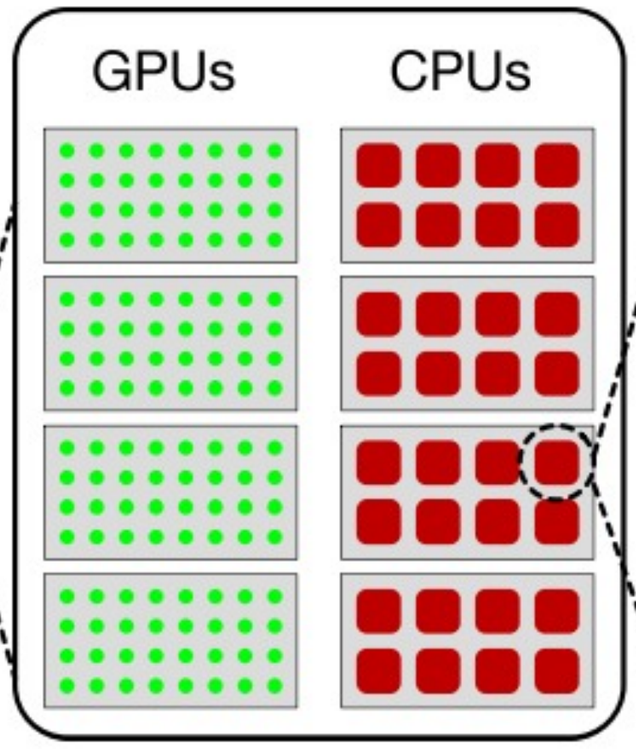
Multicore CPU + many core GPU
caches, interconnects

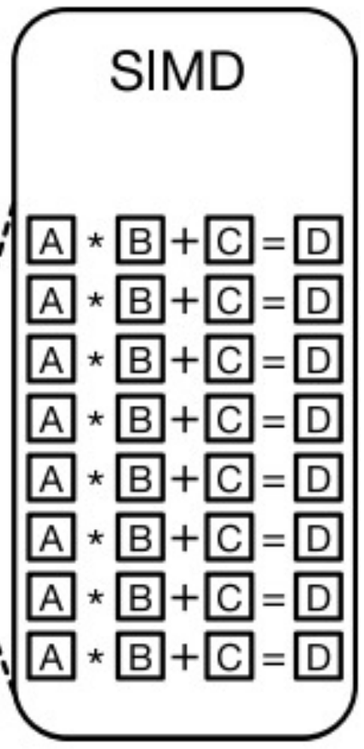
Multiple levels of **hardware parallelism**
Multiple levels of **parallelization**

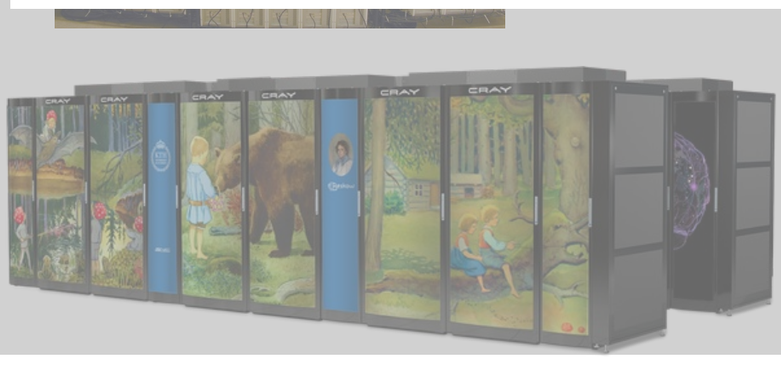


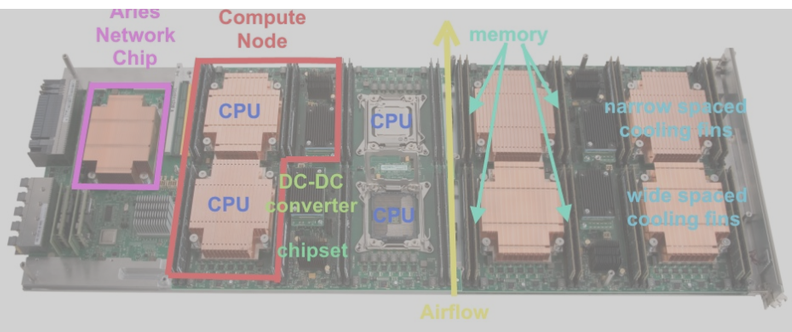


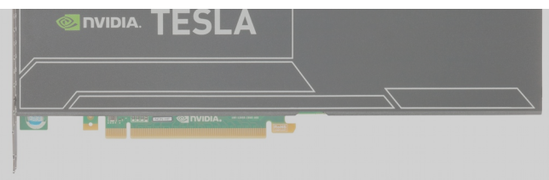












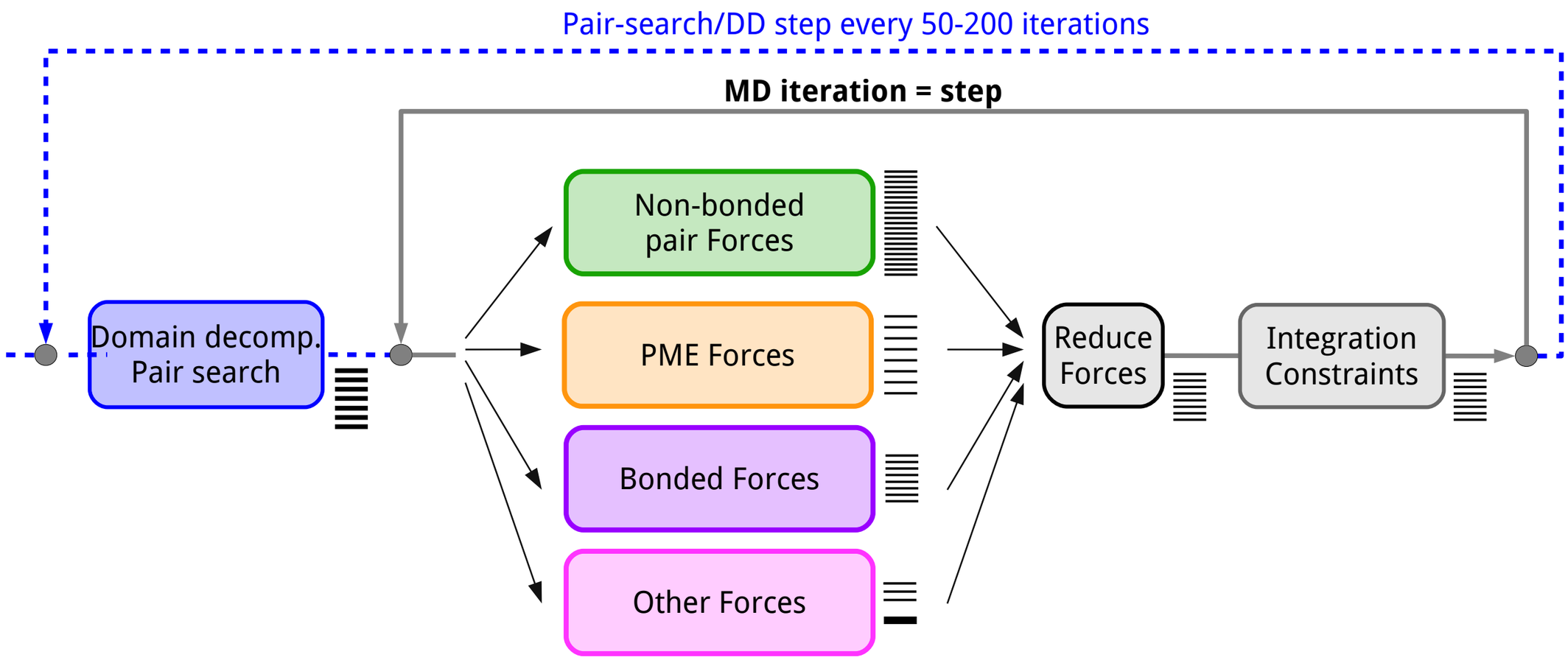
Compute cluster or cloud
Networked computers:
topology, bandwidth, latency

Compute node / workstation

NUMA topology, PCIe

Multicore CPU + many core GPU
caches, interconnects

Concurrency within an the MD step



Decomposition approaches

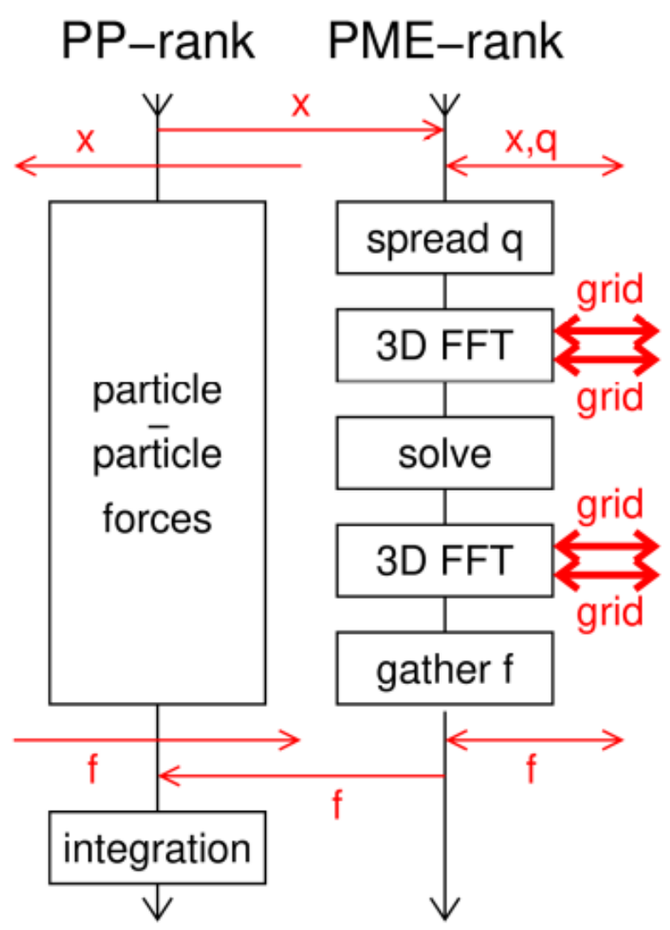
- Problem decomposition approaches:
  - single-trajectory
  - multi-trajectory: ensemble / workflows

- Work decomposition within a simulation:
  - data:
    - spatial decomp (eighth shell)
    - force decomp (intra-domain)
  - task decomposition
    - async force offload
    - MPMD to reduce 3D-FFT communication

GROMACS parallelization

Parallelism exploited on **multiple levels**:

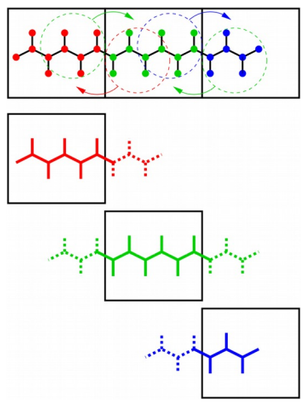
SIMD / threading / NUMA / async offload / MPI

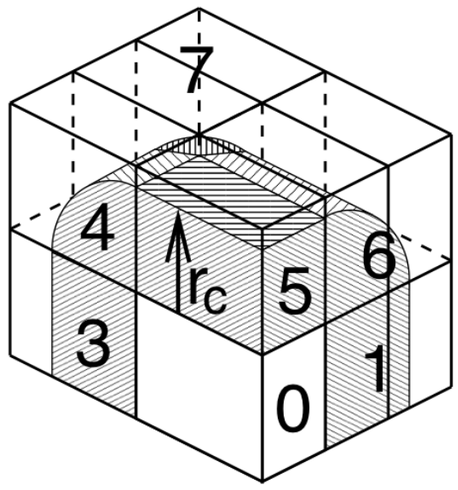


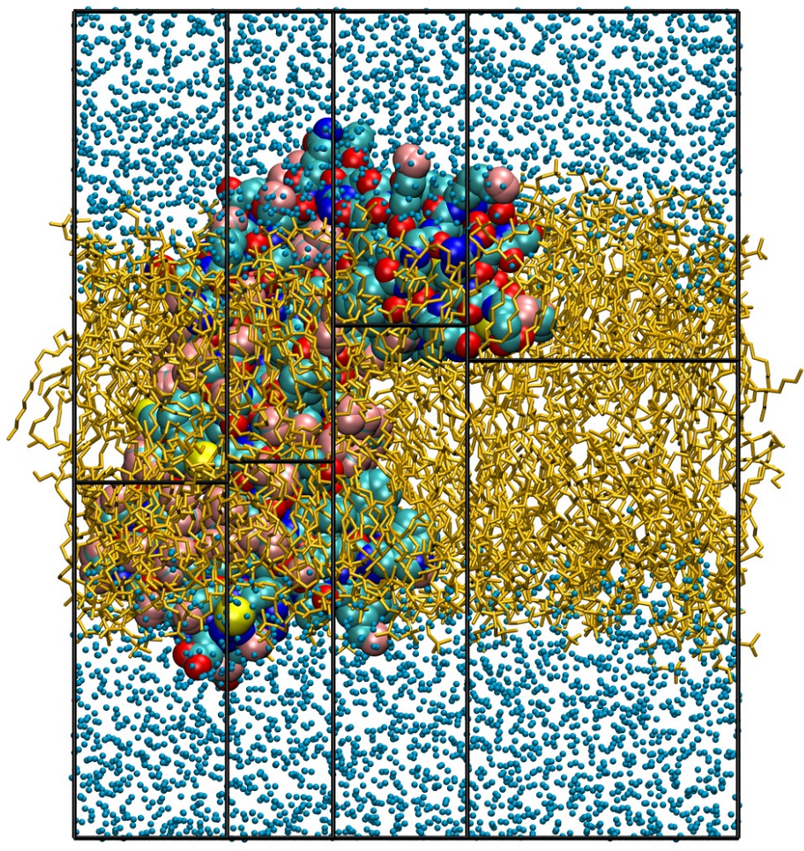
- **Hierarchical** parallelization**:**

  target each level of hw parallelism

  - MPI: SPMD / MPMD; thread-MPI
  - OpenMP
  - SIMD: 14 flavors (SIMD library abstraction)
  - CUDA, OpenCL



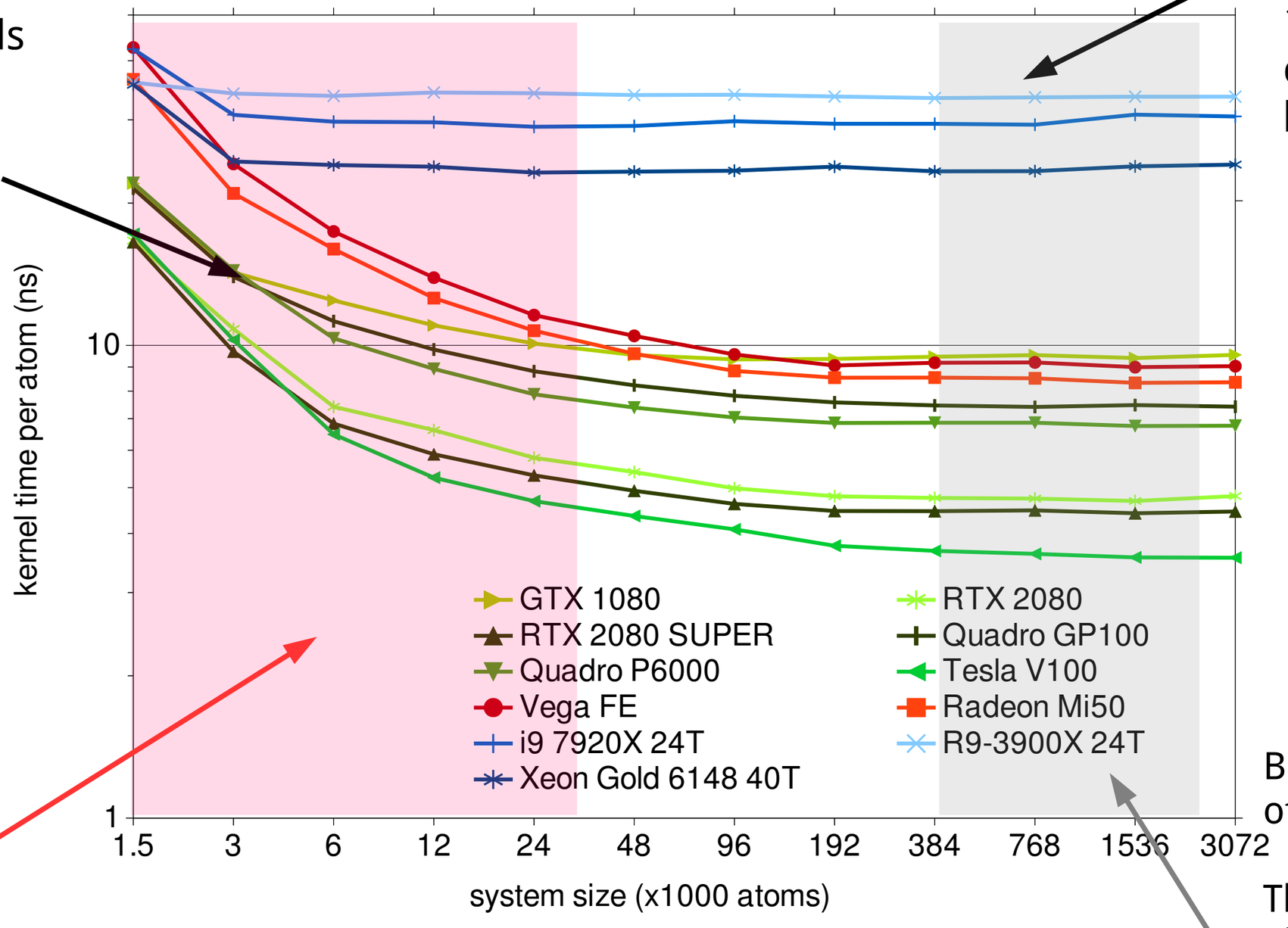




Pair interaction kernel throughput

GPUs very sensitive to input size: fixed overheads kernel startup SM load imabalance

CPUs insensitive to input size to 100s atoms/core cache effects at large inputs



Strong scaling regime: **where most of our efforts go!**

Benchmark "show-off" regime:

This is where the **"free lunch" from new hardware** comes in full effect

GROMACS Heterogeneous GPU offload

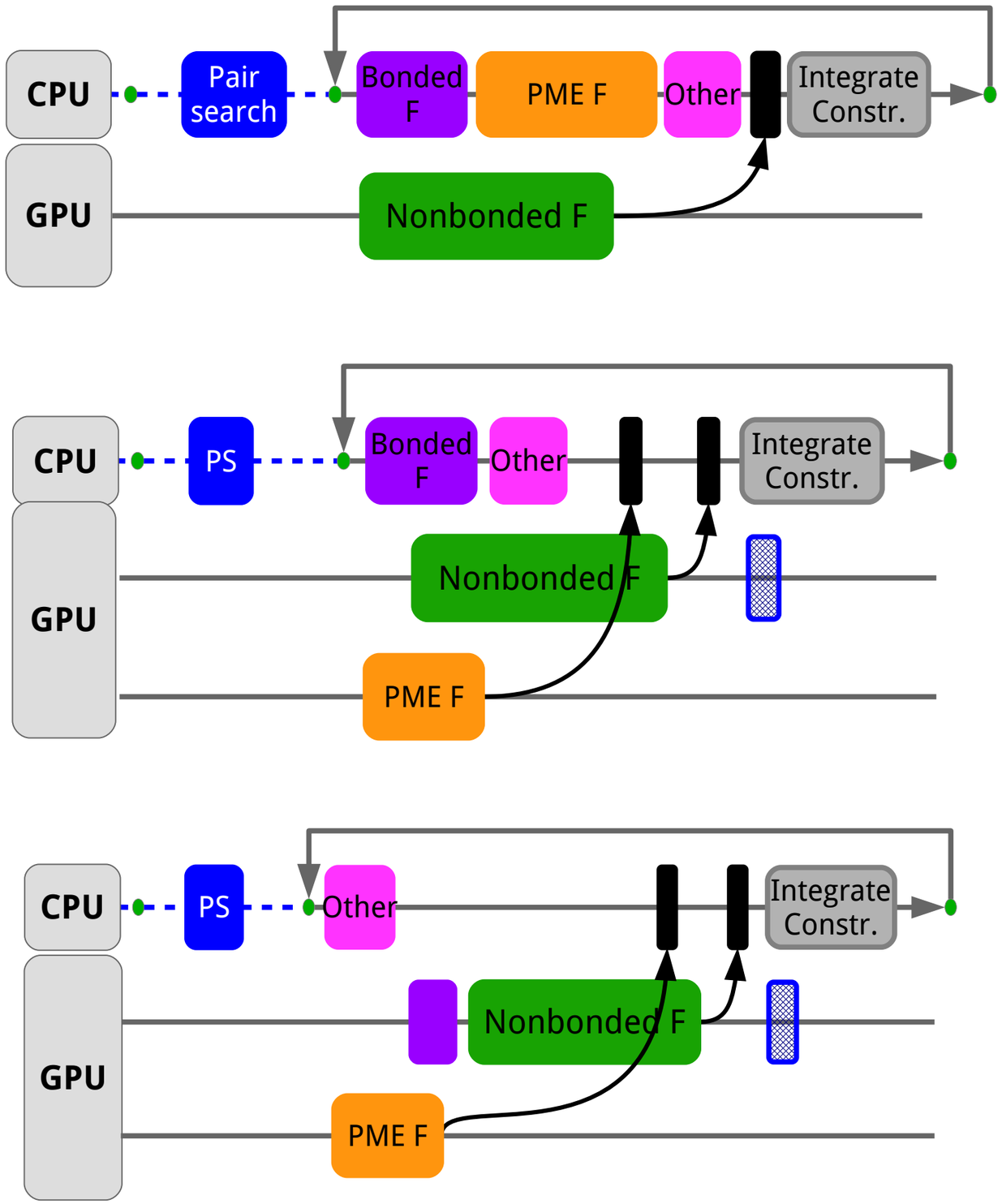
- **Maintains the versatility** of GROMACS
  - the majority of the features supported
  - “full port” to multiple toolkits/APIs not an option for a large codebase (& small team)

- **Performance**
  - use CPU & GPU for the tasks they are best at
  - flexibility for performance: adapt to CPU/GPU hw balance

- **Portability and hardware support:**
  - CUDA, OpenCL, SYCL
  - NVIDIA, AMD, Intel hardware support

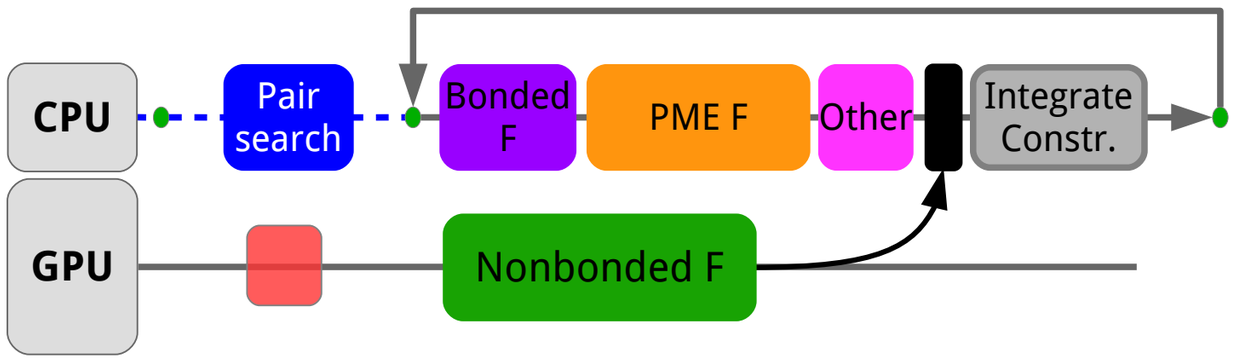
- Challenges:
  - flexibility vs complexity
  - fast CPU code, so it is often worth using
  - short time/step:
    - at peak: 200-500 us/iteration at peak (with 20-40 compute tasks/iteration)
    - latency matters

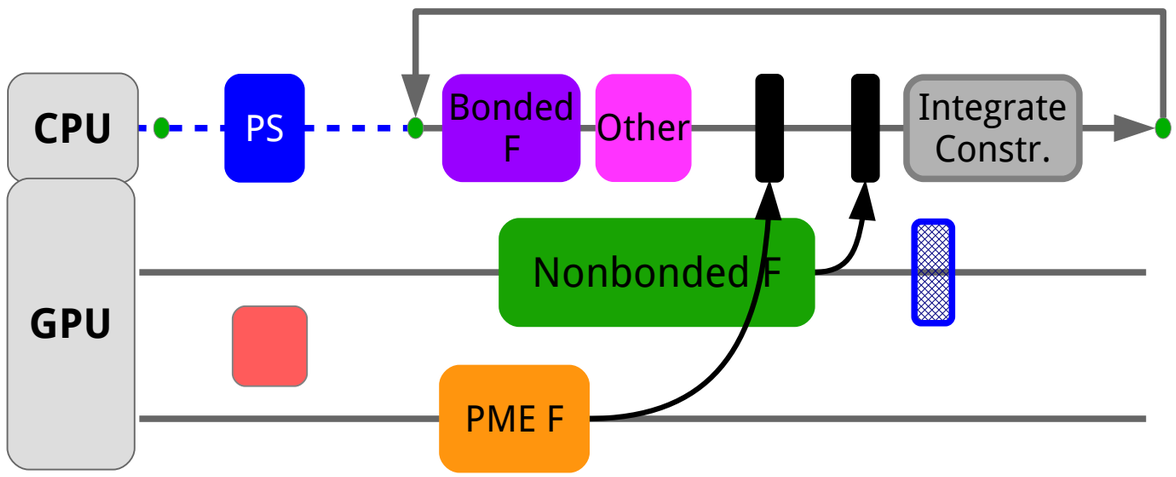
Force offload schemes

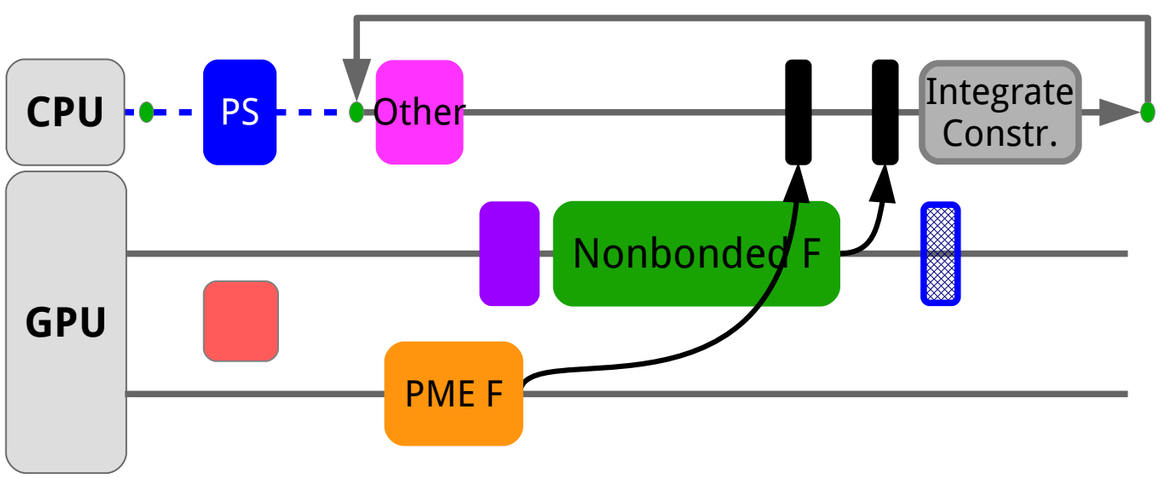


- Offloading different force components allows adjusting to hardware balance

Force offload schemes





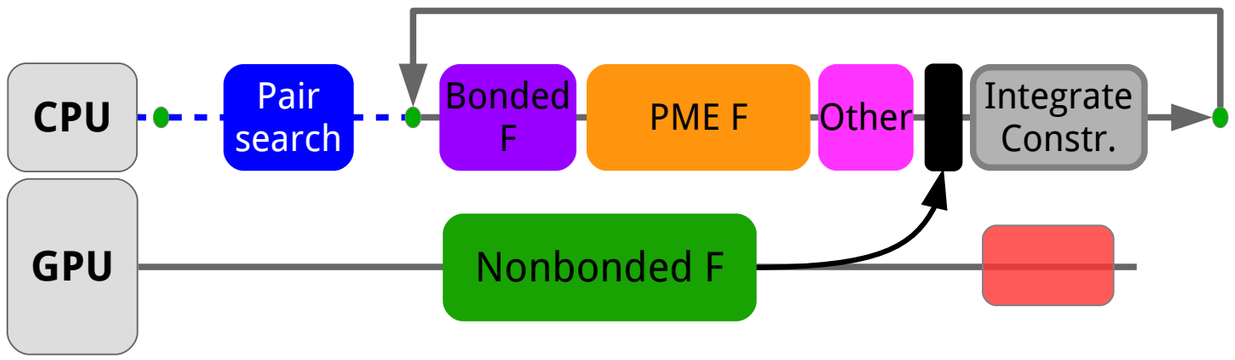


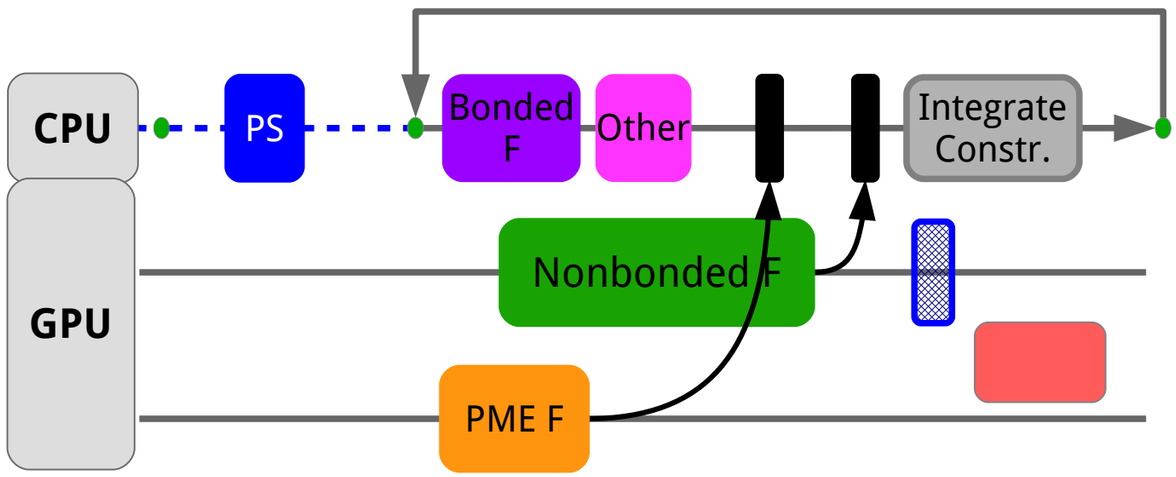
- Offloading different force components allows adjusting to hardware balance

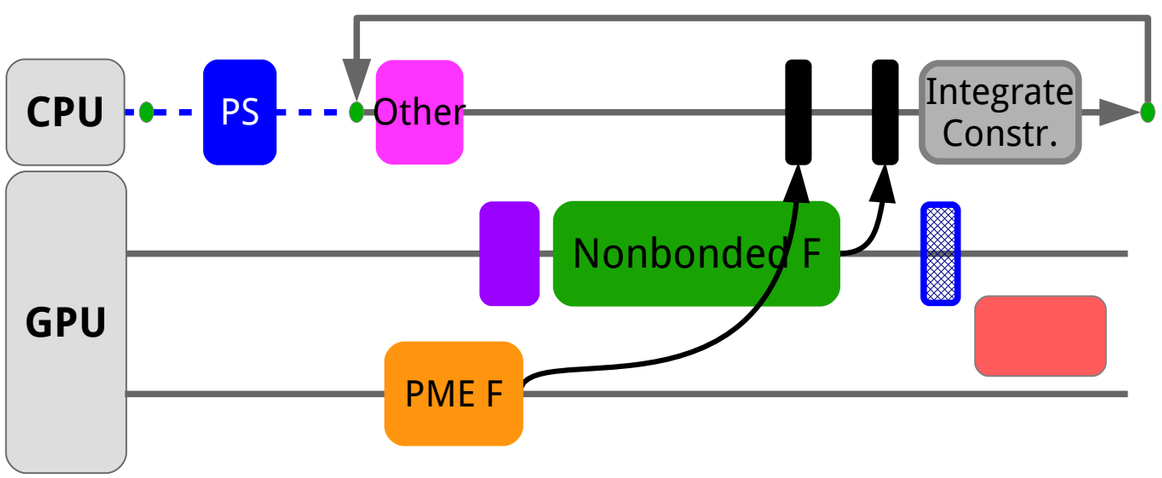
- Pair seach / DD:
  - complex code kept on CPUs

    → use algorithmic optimization to improve CPU—GPU overlap & reduce GPU idle-time

Force offload schemes







Integration on the CPU

=>

CPU – GPU data movement needed

Amdahl's law:

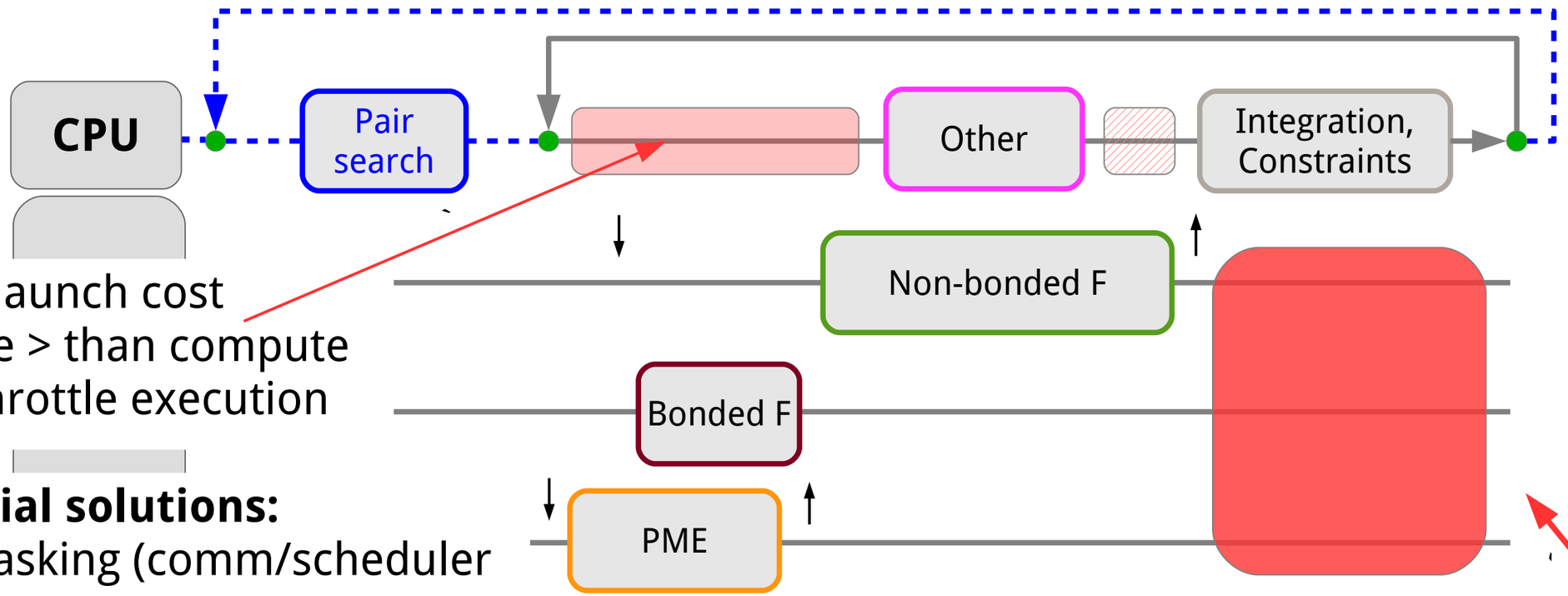
GPUs get faster ,

CPU integration time increases

- Solutions:
  - use force decomp & pipeline update (PCIe bottleneck!)
  - offload integration

GPU offload: challenges

- Increasing % of wall-time in integration/constraints: **GPU left idle**
  - just offload to GPU?
    - Pros: good for very dense GPU setups / fast accelerators
    - Cons: more GPU code to maintain, often won't actually be faster
  - allow CPU-GPU work to overlap during update too
    - Pros: universal, widely useful (CPU-only too), less porting work, makes use of CPUs
    - Cons: might not reach the perfect overlap in some cases



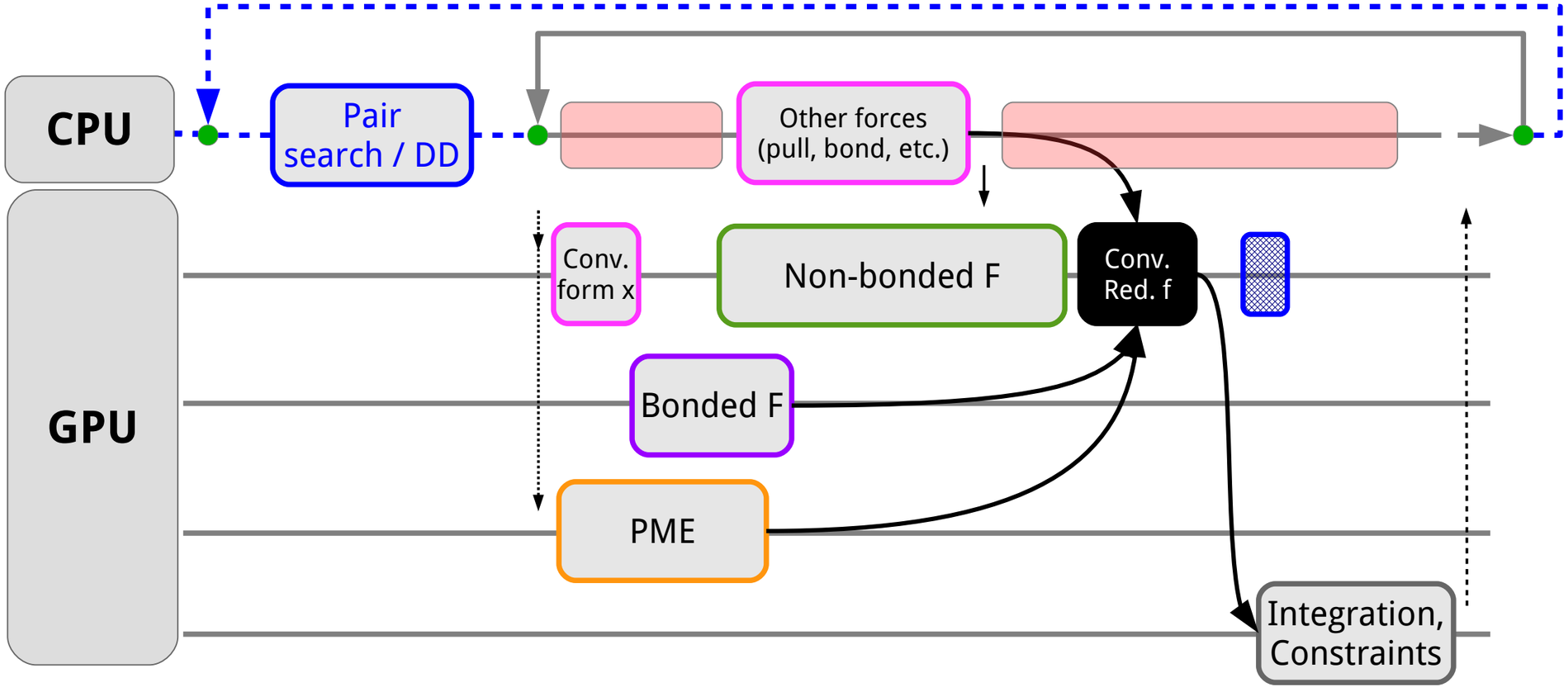
Kernel launch cost
- can be > than compute
- can throttle execution

Potential solutions:
- CPU tasking (comm/scheduler thread)
- **CUDA graphs**
- ~~persistent kernels~~

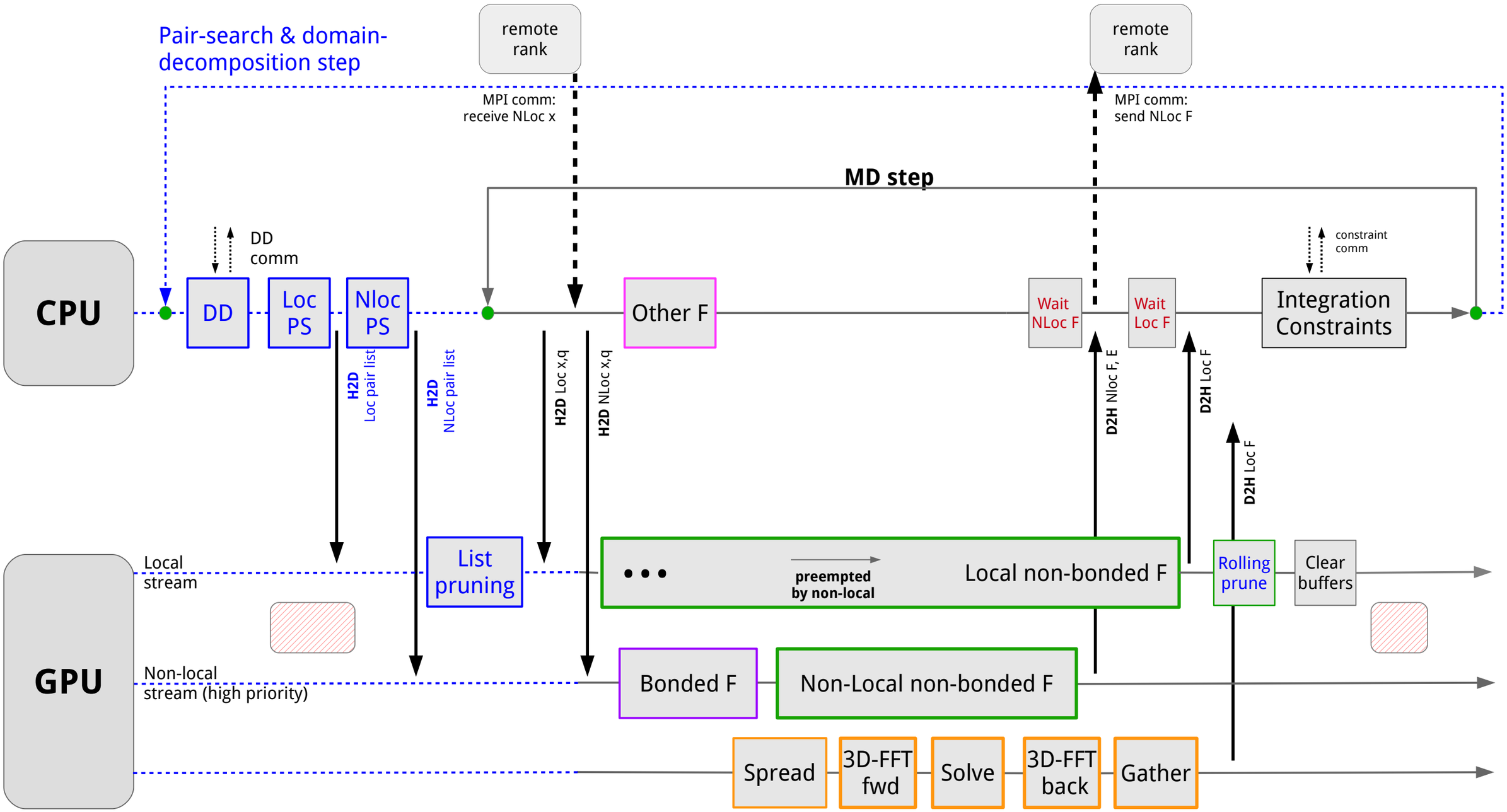
integration % increasing!
=>
GPU left idle

GPU resident MD steps

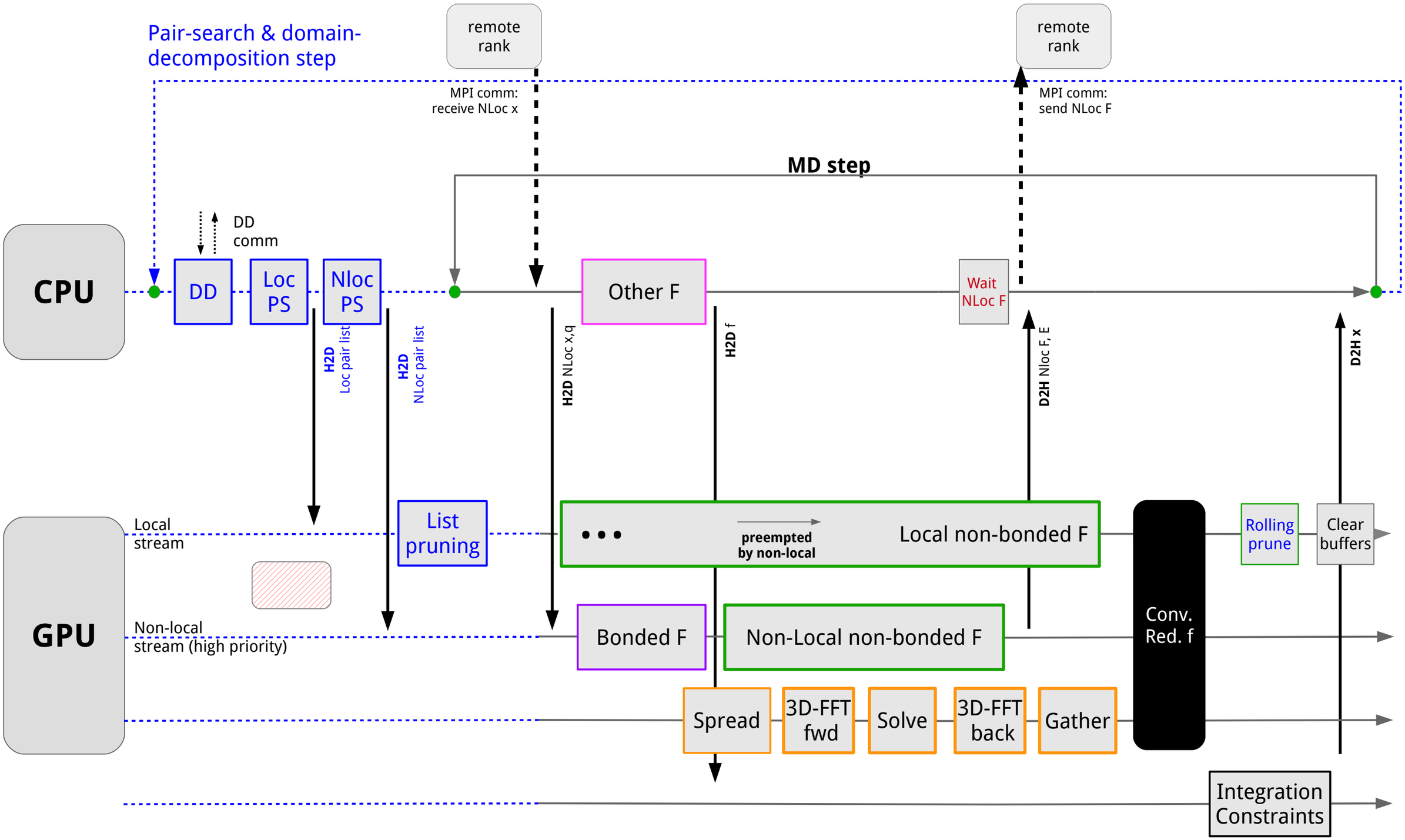
- **x/f resident on the GPU** as long as possible
- **Trade GPU idling for CPU idling**: ideal for GPU dense architectures
- CPU supporting role (“back-offload”):
  - non-offloaded per-step algorithms
  - infrequent tasks (search, DD)
- Major benefits with direct communication



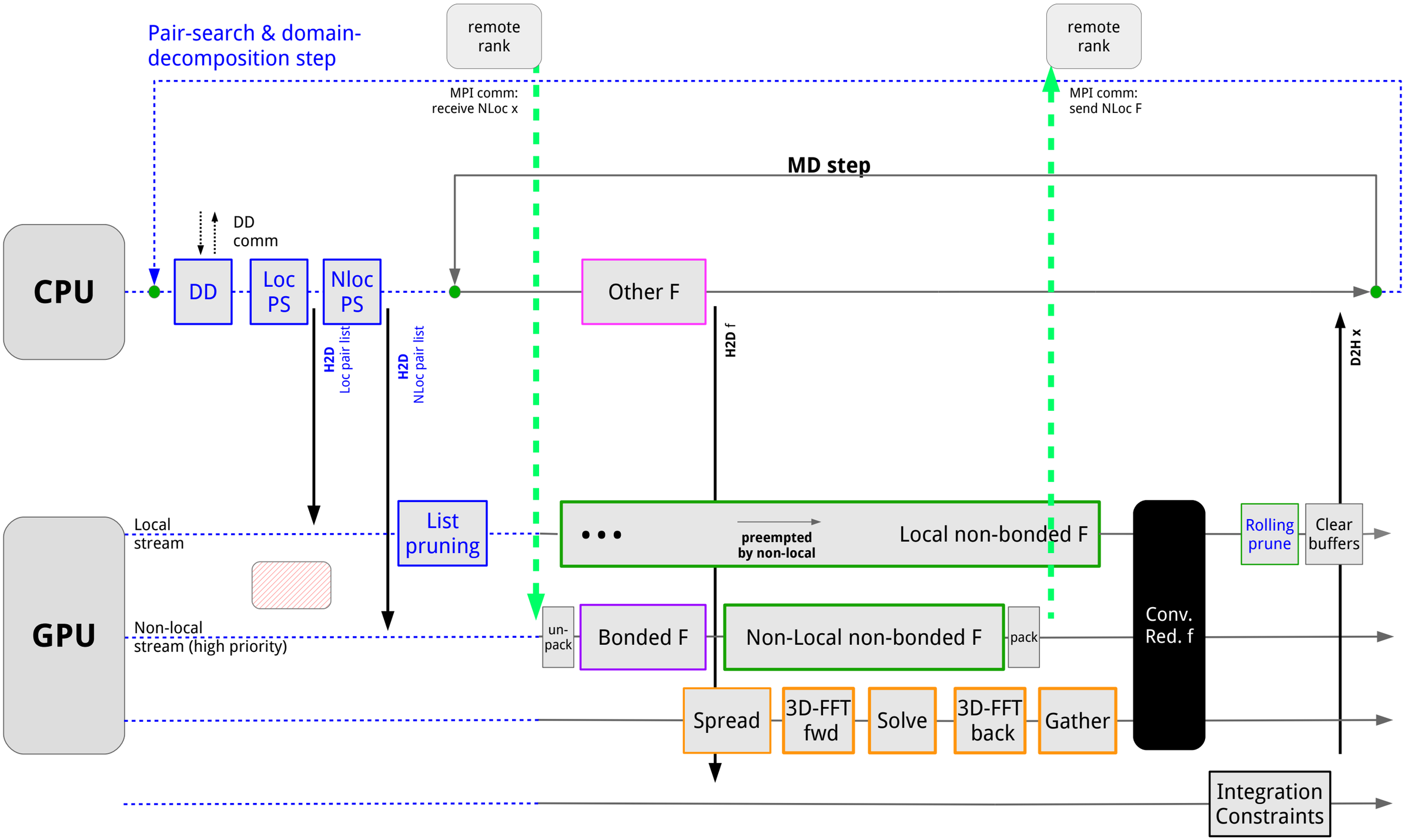
Multi-node force offload



Multi-node step offload

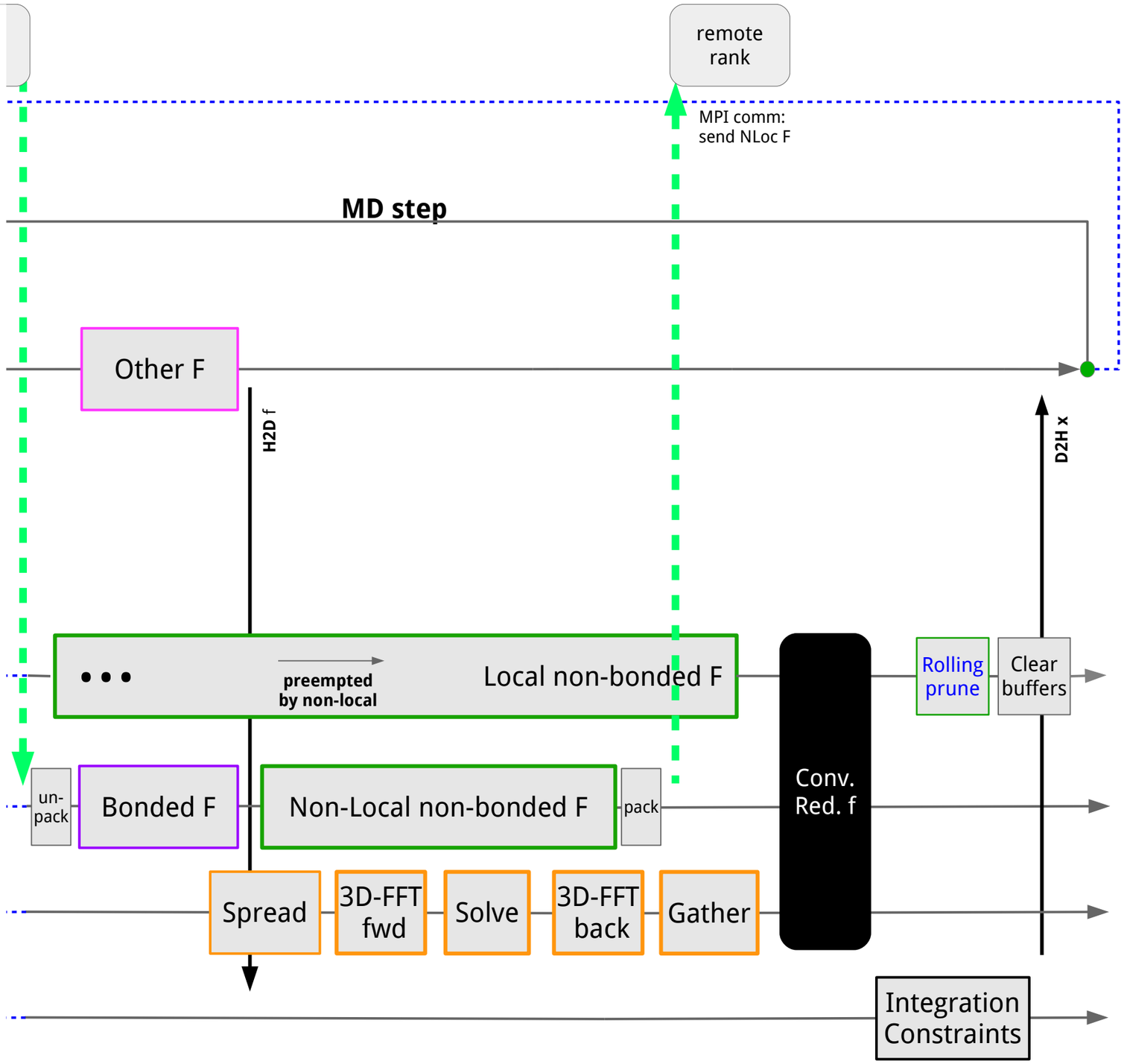


Multi-node step offload & P2P GPU comm



Multi-node step offload & P2P GPU comm

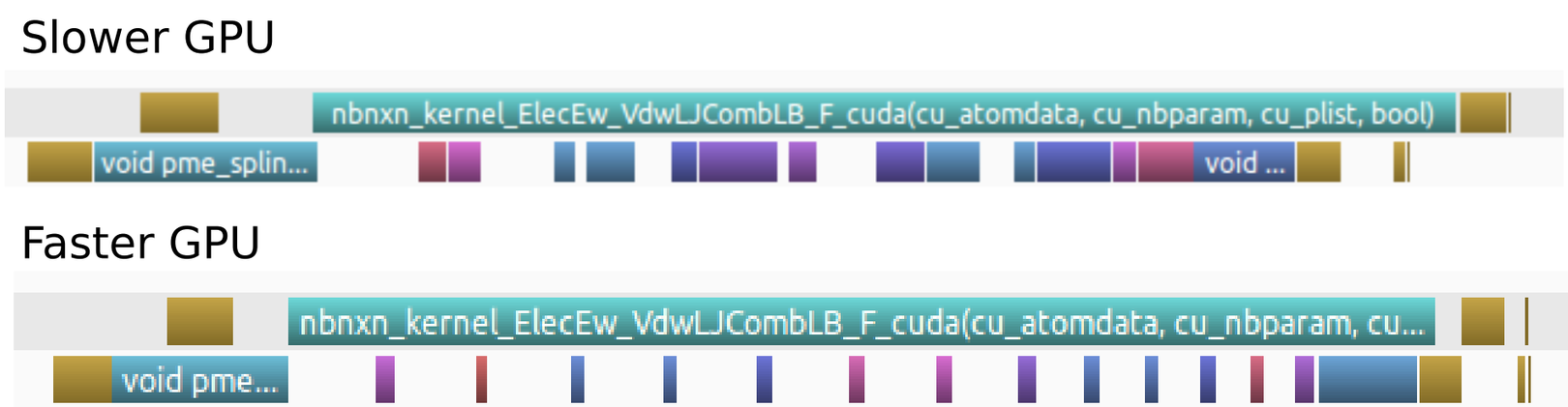
- thread-MPI allows fully async comm
- Challenges:
  - MPI is not ideal – does not allow fully async tasks
  - wasting all CPU cores of a rank before MPI
  - need comm thread specialization to conserve CPU for “other F”
  - notified receiver



Critical path optimization challenges

- Forward progress is not ensured by priority
  - eager execution fills the GPU
  - low-prio kernel(s) compete with high low-prio kernels
    - offloading a small task for locality can hurt performance delaying a task on the critical path
    - more priority levels may help but won't solve the issue
    - **proposed solution:** (conditionally) reserve GPU SMs for some tasks

Competing high prio work and "backfill" low prio kernel

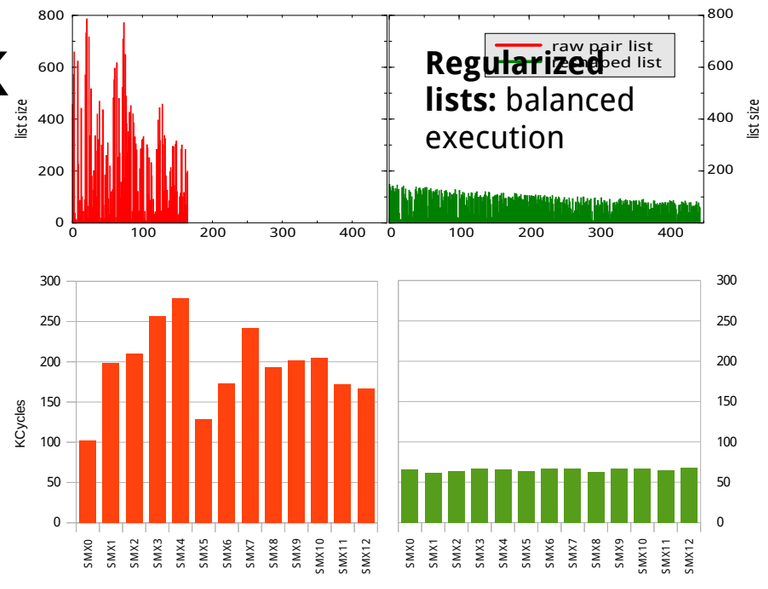


Topology and affinity challenges

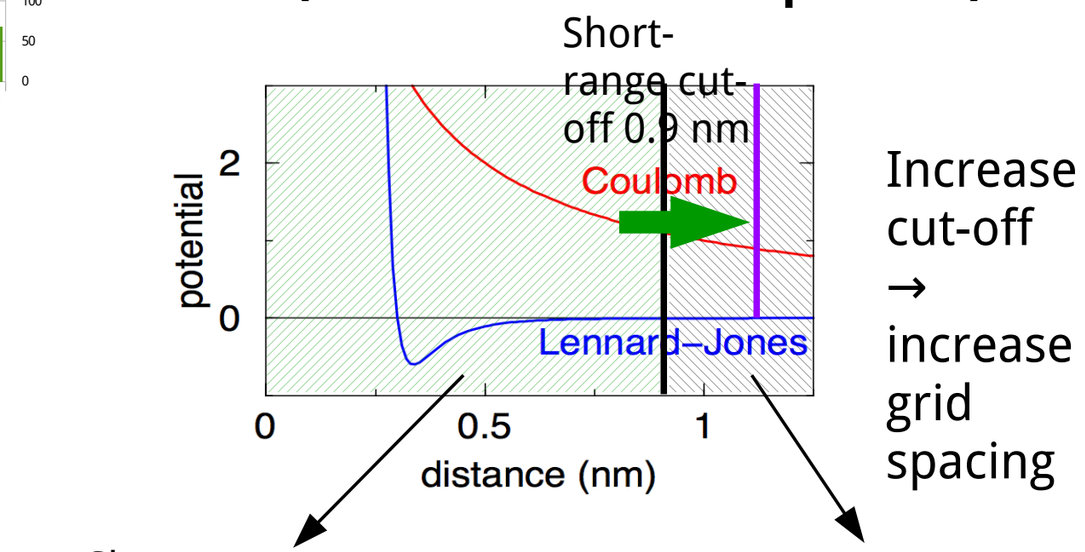
- intra-node
  - rank to GPU mapping (not implemented)
  - adapt decomposition and communication strategies to topology
- inter-node
  - network topology / node mapping
  - ensemble optimization

Multi-level load balancing

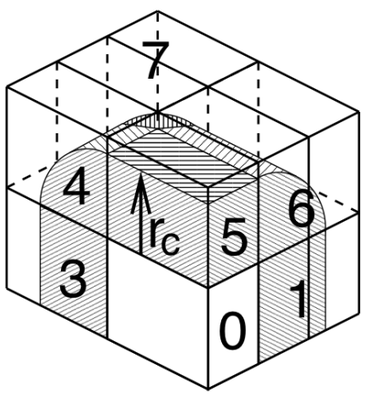
GPU intra-task balancing (offline)

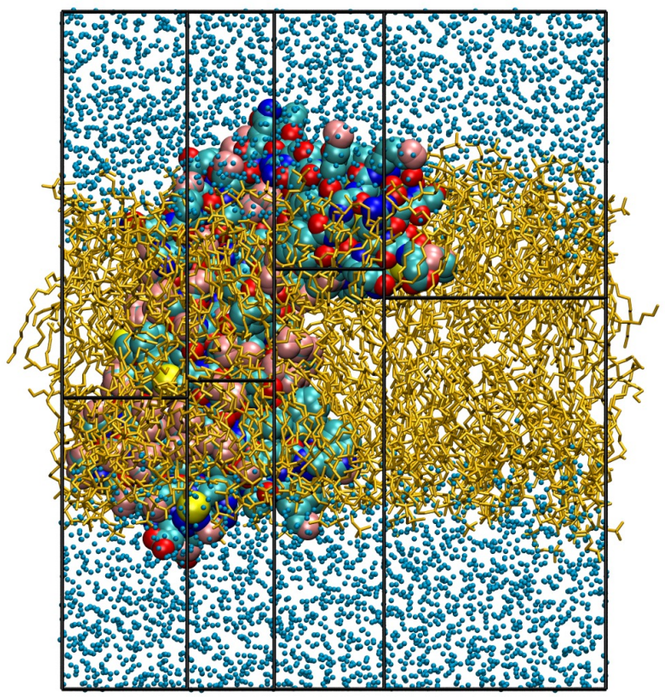


PP- PME, CPU-GPU (static / infrequent)



Eighth shell domain-decomposition & online dynamic LB





Multi-level load balancing

- Major load balancing challenge:
  - measuring wall-time of CPU+GPU work is not possible (cudaEvents)
  - need new model to estimate work based on other metrics (flops + total time of a DD range)

dynamic LB