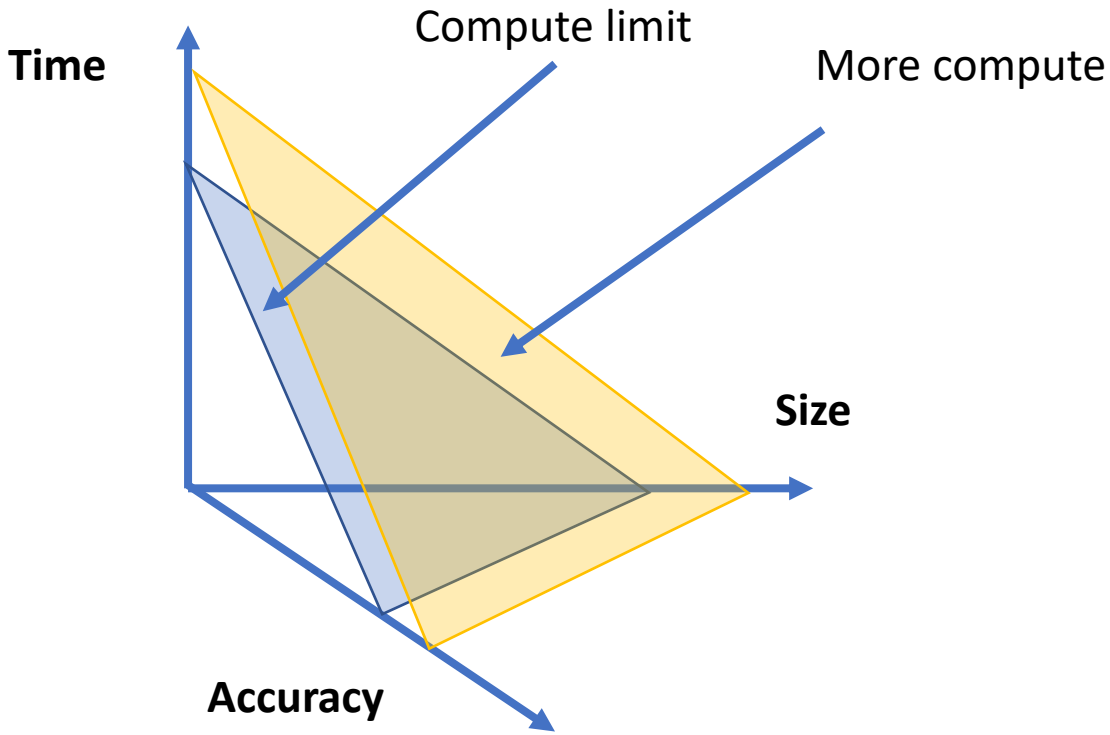
Los Alamos
NATIONAL LABORATORY

Molecular dynamics on exascale computers: a case study

Danny Perez
Theoretical Division T-1

What do we *want* out of exascale: more compute = more science



Plan

- A case study of MD on exascale computers with the SNAP potential
  - Porting SNAP to GPUs
  - Parallel MD using SNAP: weak and strong scaling

- The timescale problem of MD
  - Parallelizing over time instead of space with Parallel Trajectory Splicing

- Accuracy tradeoffs

Case study

Billion atom molecular dynamics simulations of carbon at extreme conditions and experimental time and length scales

Kien Nguyen-Cong*
nguyencong@usf.edu
University of South Florida
Tampa, FL, USA

Jonathan T. Willman*
jwillma2@usf.edu
University of South Florida
Tampa, FL, USA

Stan G. Moore
stamoor@sandia.gov
Sandia National Laboratories
Albuquerque, NM, USA

Anatoly B. Belonoshko
anatoly@kth.se
Royal Institute of Technology (KTH)
Stockholm, Sweden

Rahulkumar Gayatri
rgayatri@lbl.gov
NERSC
Berkeley, CA, USA

Evan Weinberg
eweinberg@nvidia.com
NVIDIA Corporation
Santa Clara, CA, USA

Mitchell A. Wood
mitwood@sandia.gov
Sandia National Laboratories
Albuquerque, NM, USA

Aidan P. Thompson
athomps@sandia.gov
Sandia National Laboratories
Albuquerque, NM, USA

Ivan I. Oleynik
oleynik@usf.edu
University of South Florida
Tampa, FL, USA

ABSTRACT

Billion atom molecular dynamics (MD) using quantum-accurate machine-learning Spectral Neighbor Analysis Potential (SNAP) observed long-sought high pressure BC8 phase of carbon at extreme pressure (12 Mbar) and temperature (5,000 K). 24-hour, 4650 node production simulation on OLCF Summit demonstrated an unprecedented scaling and unmatched real-world performance of SNAP MD while sampling 1 nanosecond of physical time. Efficient implementation of SNAP force kernel in LAMMPS using the Kokkos CUDA backend on NVIDIA GPUs combined with excellent strong scaling (better than 97% parallel efficiency) enabled a peak computing rate of 50.0 PFLOPs (24.9% of theoretical peak) for a 20 billion atom MD simulation on the full Summit machine (27,900 GPUs). The peak MD performance of 6.21 Matom-steps/node-s is 22.9 times greater than a previous record for quantum-accurate MD. Near perfect weak scaling of SNAP MD highlights its excellent potential to advance the frontier of quantum-accurate MD to trillion atom simulations on upcoming exascale platforms.

The SNAP method

$$\mathbf{U}_j = \sum_{r_{ik}<R_{\rm cut}} f_c(r_{ik})\mathbf{u}_j(\theta_0,\theta,\phi)$$

”Fourier” coefficients of the *local* atomic density.

J is the order of expansion.

$$\begin{aligned} B_{j_1 j_2 j} &= \mathbf{U}_{j_1} \otimes^{j}_{j_1 j_2} \mathbf{U}_{j_2} : \mathbf{U}^*_j \\ &= \mathbf{Z}^{j}_{j_1 j_2} : \mathbf{U}^*_j \end{aligned}$$

Symmetrized coefficients (rotation invariant)

$$E_i(\mathbf{B}) = \sum_{l=1}^{N_B} \beta_l B_l$$

$$\mathbf{F}_k = -\sum_{i=1}^{N}\sum_{l=1}^{N_B} \beta_l \frac{\partial B_l}{\partial \mathbf{r}_k}$$

Energy is a sum of per-atom energies, which are assumed linear in **B.** $N_B \sim J^3$

$$\begin{aligned} \frac{\partial B_{j_1 j_2 j}}{\partial \mathbf{r}_k} &= \mathbf{Z}^{j}_{j_1 j_2} : \frac{\partial \mathbf{U}^*_j}{\partial \mathbf{r}_k} \\ &\quad + \mathbf{Z}^{j_1}_{j j_2} : \frac{\partial \mathbf{U}^*_{j_1}}{\partial \mathbf{r}_k} + \mathbf{Z}^{j_2}_{j j_1} : \frac{\partial \mathbf{U}^*_{j_2}}{\partial \mathbf{r}_k} \end{aligned}$$

Forces obtained via chain rule

Vanilla SNAP code

Complexity (per atom)

```
for(int natom=0; natom<num_atoms; ++natom)
{
  // build neighbor-list for each atom
```

O(1)

$\mathbf{U}_j = \sum_{r_{ik}<}$

$B_{j_1 j_2 j} =$

$=$

$\frac{\partial B_{j_1 j_2 j}}{\partial \mathbf{r}_k} =$

$\mathbf{F}_k = -\sum_{i=1}^{N}\sum_{l=1}^{N_B}\beta_l \frac{\partial B_l}{\partial \mathbf{r}_k}$

N_nbor)

nbor)

nbor)

nbor)

Key point: modern ML potentials are complex

Listing 1: SNAP code

If you don’t run efficiently on 1 node, you won’t run efficiently on 10,000 nodes.

MD on GPUs

- Most large-scale machines rely on GPUs to provide the majority of their computing power.
- Good GPU performance is essential!

- Unfortunately, achieving high GPU performance is not easy, especially for SNAP:
  - Deeply nested loops
  - Loop structure not regular
  - Loops are “narrow”

NOVEMBER 2022

1	Frontier	HPE Cray EX235a, AMD Opt 3rd Gen EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-10
2	Fugaku	Fujitsu A64FX (48C, 2.2GHz), Tofu Interconnect D
3	LUMI	HPE Cray EX235a, AMD Opt 3rd Gen EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-10
4	Leonardo	Atos Bullsequana inteXeon (32C, 2.6 GHz), NVIDIA A100 quad-rail NVIDIA HDR100 Infiniband
5	Summit	IBM POWER9 (22C, 3.07GHz), NVIDIA Volta GV100 (80C), Dual-Rail Mellanox EDR Infiniband

Original SNAP implementation circa 2012

- Christian Trott (SNL) ported the LAMMPS SNAP C++ code to Kokkos in ExaMiniMD (**proxy app**), then ported to Kokkos LAMMPS by Stan Moore (SNL)
- Used advanced Kokkos features: hierarchical parallelism and scratch memory, unsure how to get better performance at the time
- Still: depressing fraction of peak on GPU compared to CPU

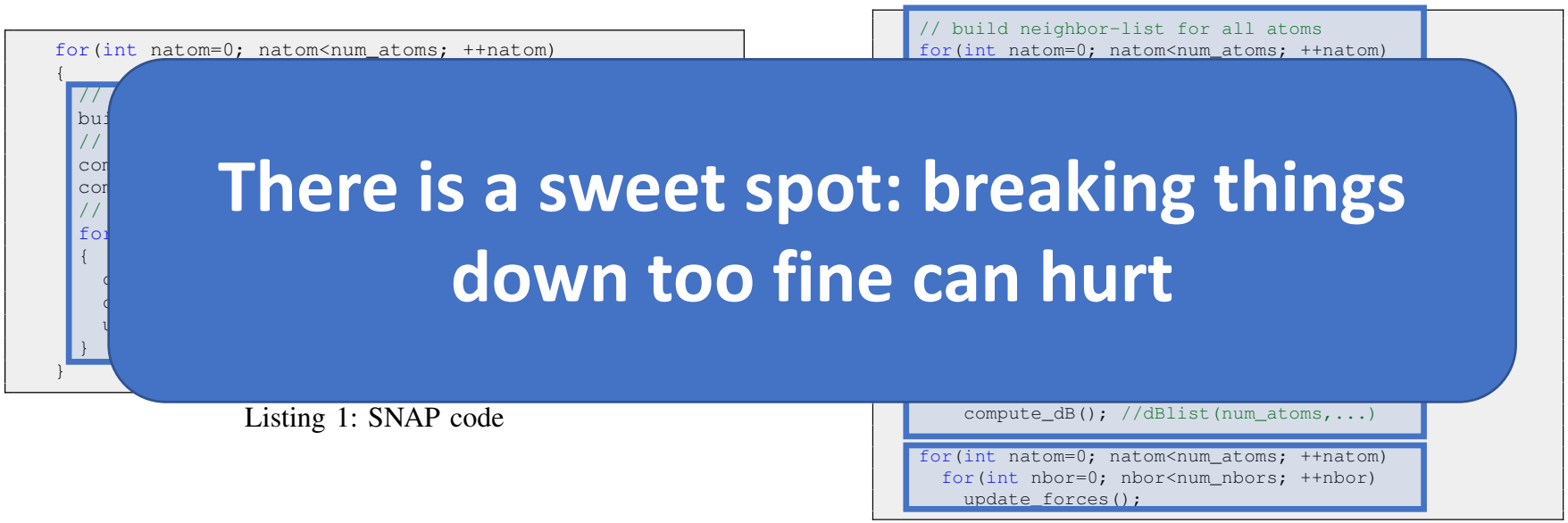
Node	Machine	Year	SNAP (Katom-steps/s)	Peak/node (Tflops)	Fraction-of-Peak (normalized)
IBM PowerPC	Mira (ANL)	2012	2.52	0.205	1.0
Intel SandyBridge	Chama (SNL)	2012	17.7	0.332	4.34
AMD CPU	Titan (ORNL)	2013	5.35	0.141	3.09
NVIDIA K20X	Titan (ORNL)	2013	2.60	1.31	0.161
Intel Haswell	Cori (NERSC)	2016	29.4	1.18	2.03
Intel KNL	Cori (NERSC)	2016	11.1	2.61	0.346
NVIDIA P100	Ride (SNL)	2016	21.8	5.30	0.335
Intel Broadwell	Serrano (SNL)	2017	25.4	1.21	1.71

No silver bullet

- **Adjoint refactor**: algorithmic redesign that reduced the computational complexity and memory footprint by large factor
- **Flattened jagged multi-dimensional arrays:** reduced memory use
- **Major kernel refactor:** Broke one large kernel into many smaller kernels, reordered loop structure
- **Changed the memory data layout** of an array between kernels via transpose operations
- **Refactored loop indices and data structures** to use complex numbers and multi-dimensional arrays instead of arrays of structs
- Refactored some kernels to **avoid thread atomics** and use of **global memory**
- Judiciously used **Kokkos hierarchical parallelism** and **GPU shared memory**
- **Fused** a few selected **kernels**, which helped eliminate intermediate data structures and reduced memory use
- Added an AoSoA **memory data layout** which enforced perfect coalescing and load balancing in one of the kernels
- **Symmetrized data layouts** of certain matrices, which reduced memory overhead and use of thread atomics on GPUs (also improved CPU performance)
- Large refactor of Wigner matrices + derivatives to **use AoSoA data layout**

[Gayatri et al, arXiv:2011.12875v1]

Vignette 1: loop structure matters



There is a sweet spot: breaking things down too fine can hurt

Listing 1: SNAP code

Listing 2: Refactored TestSNAP code

Ease register pressure, each kernel can be tuned separately

Increased memory usage, code complexity

Vignette 2: there is more than one way to write a loop

$$B_{j_1 j_2 j} = \mathbf{U}_{j_1} \otimes^{j}_{j_1 j_2} \mathbf{U}_{j_2} : \mathbf{U}^*_j$$

$$\mathbf{F}$$

$$\frac{\partial B_{j_1 j_2 j}}{\partial \mathbf{r}_k} = \mathbf{Z}^{j}_{j_1 j_2} : \frac{\partial \mathbf{U}_j}{\partial \mathbf{r}_k} + \mathbf{Z}^{j_1}_{j j_2} : \frac{\partial \mathbf{U}^*_{j_1}}{\partial \mathbf{r}_k} + \mathbf{Z}^{j_2}_{j j_1} : \frac{\partial \mathbf{U}^*_{j_2}}{\partial \mathbf{r}_k}$$

$$\mathbf{F}_k = -\sum_{i=1} \sum_{j=0} \mathbf{Y}_j : \frac{\partial \mathbf{U}_j}{\partial \mathbf{r}_k}.$$

Looks obvious… once someone else tells you about it.

$O(J^5 N_{nbor})$ storage for *Z*
$O(J^3)$ force calculation

$O(J^3)$ storage for Y
$O(J)$ force calculation

Memory matters

- **Adjoint refactor**: algorithmic redesign that reduced the computational complexity and memory footprint by large factor
- **Flattened jagged multi-dimensional arrays:** reduced memory use
- **Major kernel refactor:** Broke one large kernel into many smaller kernels, reordered loop structure
- **Changed the memory data layout** of an array between kernels via transpose operations
- **Refactored loop indices and data structures** to use complex numbers and multi-dimensional arrays instead of arrays of structs
- Refactored some kernels to **avoid thread atomics** and use of **global memory**
- Judiciously used **Kokkos hierarchical parallelism** and **GPU shared memory**
- **Fused** a few selected **kernels**, which helped eliminate intermediate data structures and reduced memory use
- Added an AoSoA **memory data layout** which enforced perfect coalescing and load balancing in one of the kernels
- **Symmetrized data layouts** of certain matrices, which reduced memory overhead and use of thread atomics on GPUs (also improved CPU performance)
- Large refactor of Wigner matrices + derivatives to **use AoSoA data layout**

[Gayatri et al, arXiv:2011.12875v1]

No silver bullet

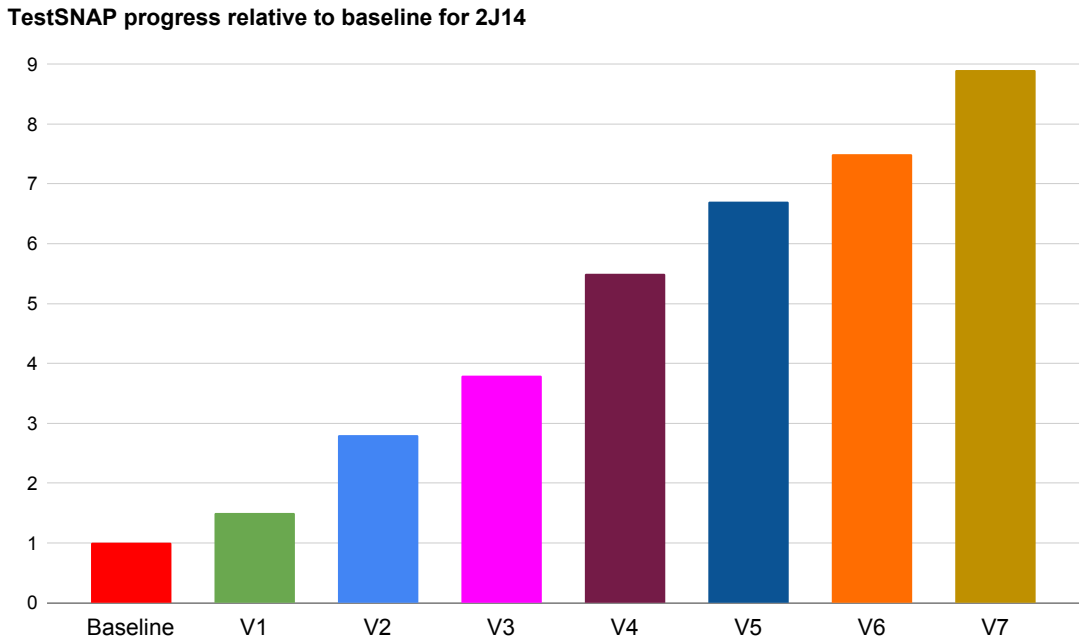
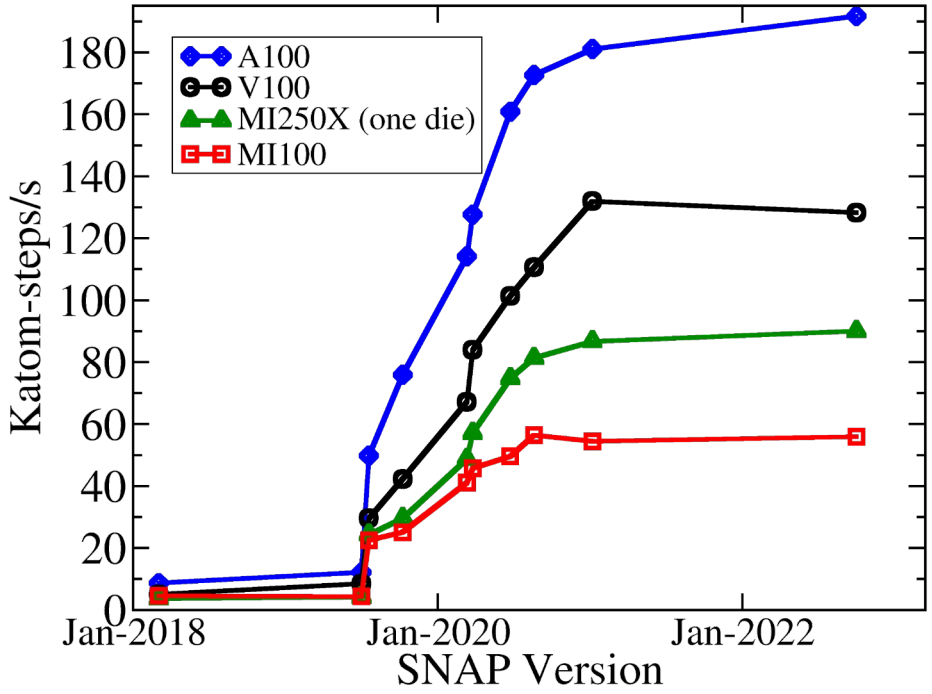


Fig. 3: TestSNAP progress relative to baseline for 2J14 problem size on NVIDIA V100.

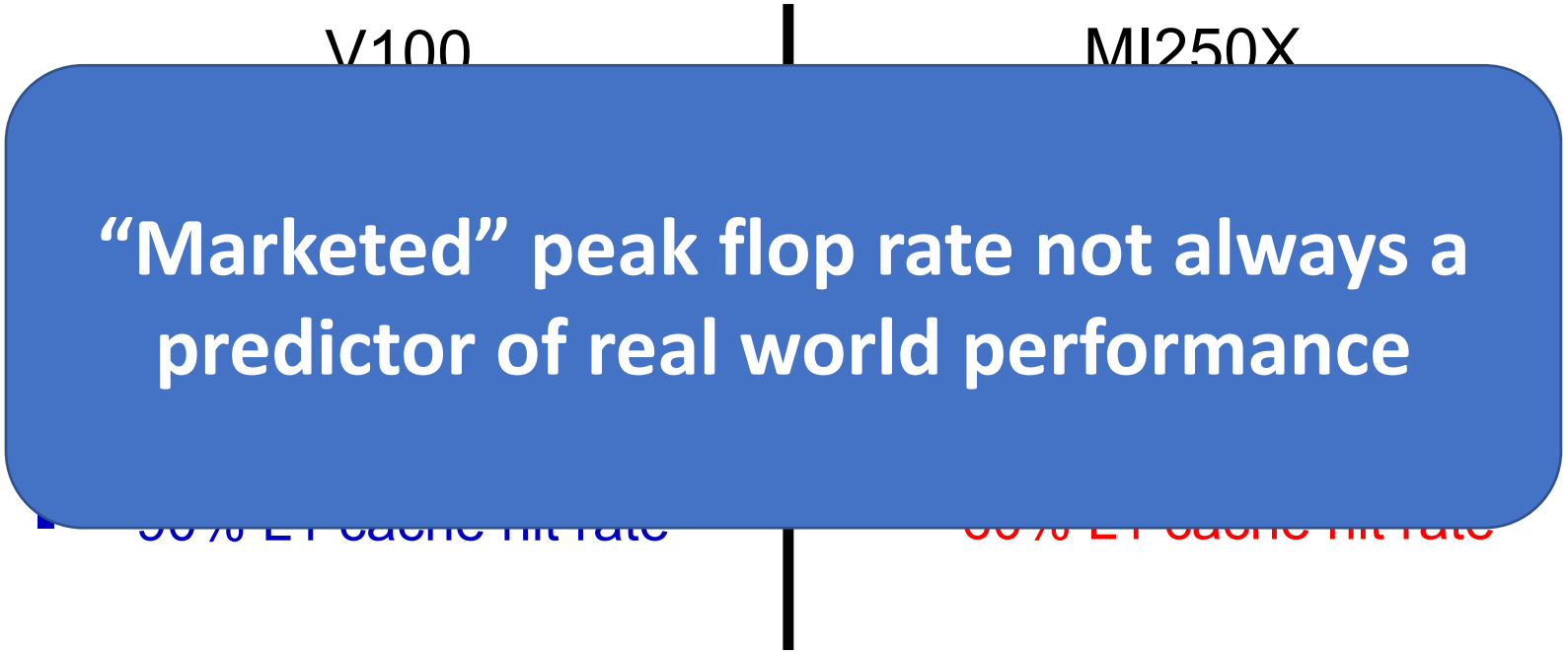
Some ideas are transferable, but…



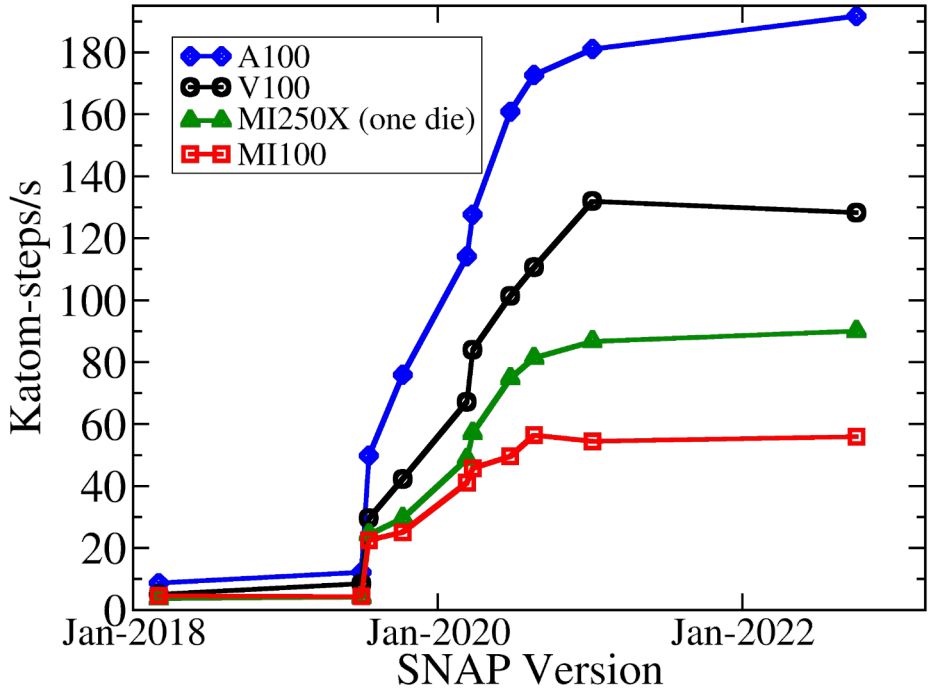
- Optimization targeted NVIDIA/V100
- **26x performance improvement for V100**
- Almost all optimization improved performance on AMD/MI250X also
- **24x for MI250X**

- **MI250X/V100 FLOPS = 2.4x**
- **MI250X/V100 SNAP = ~0.7x**
- **We lost over 3x performance/FLOPS, why?**

The devil is in the details



Time is money



- New ML approaches for potentials are constantly being proposed
- We cannot afford to spend years optimizing each
- We either need to:
  - Become dramatically better at this
  - Down-select to a few forms that are worth investing in
  - Teach the machines to optimize themselves

Algorithmics are also important!

- Very low-level optimization are crucial

- Mathematics can certainly help here, but probably won't be the main focus on this program until new ideas are fleshed up and ready to be implemented (e.g., around the Hackathon)

- Perhaps of more immediate interest: high-level algorithmics choices also make a huge difference, especially for parallelization schemes: mathematics and domain knowledge are critical there

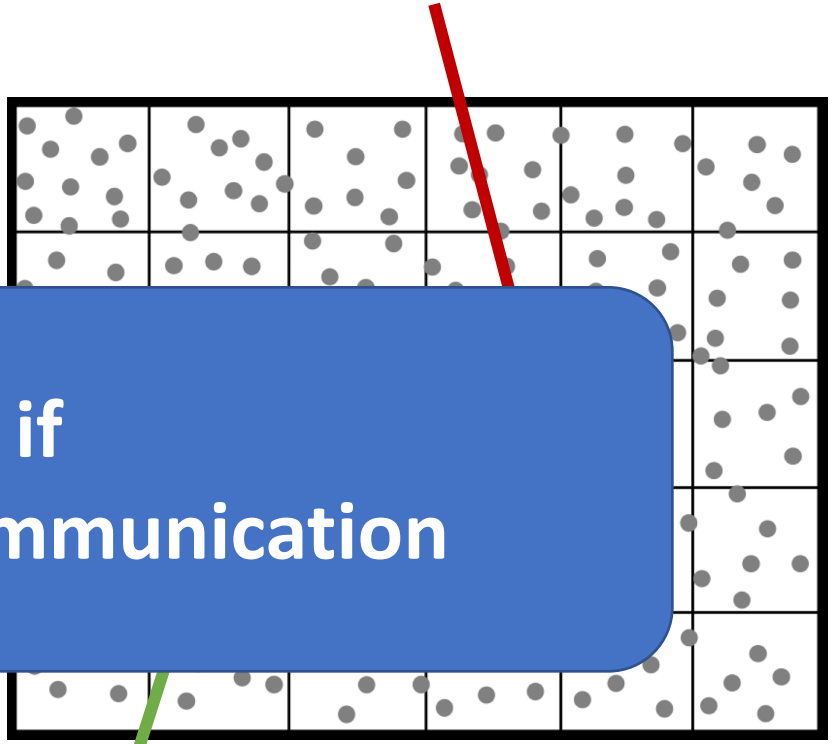
Parallel MD

Most cycles spent here

Get forces $\boldsymbol{F} = -\nabla V(\boldsymbol{r}^{(i)})$ and $\boldsymbol{a} = \boldsymbol{F}/m$

Move atoms: $\boldsymbol{r}^{(i+1)} = \boldsymbol{r}^{(i)} + \boldsymbol{v}^{(i)}\Delta t + 1/2\ \boldsymbol{a}\Delta t^2 +$

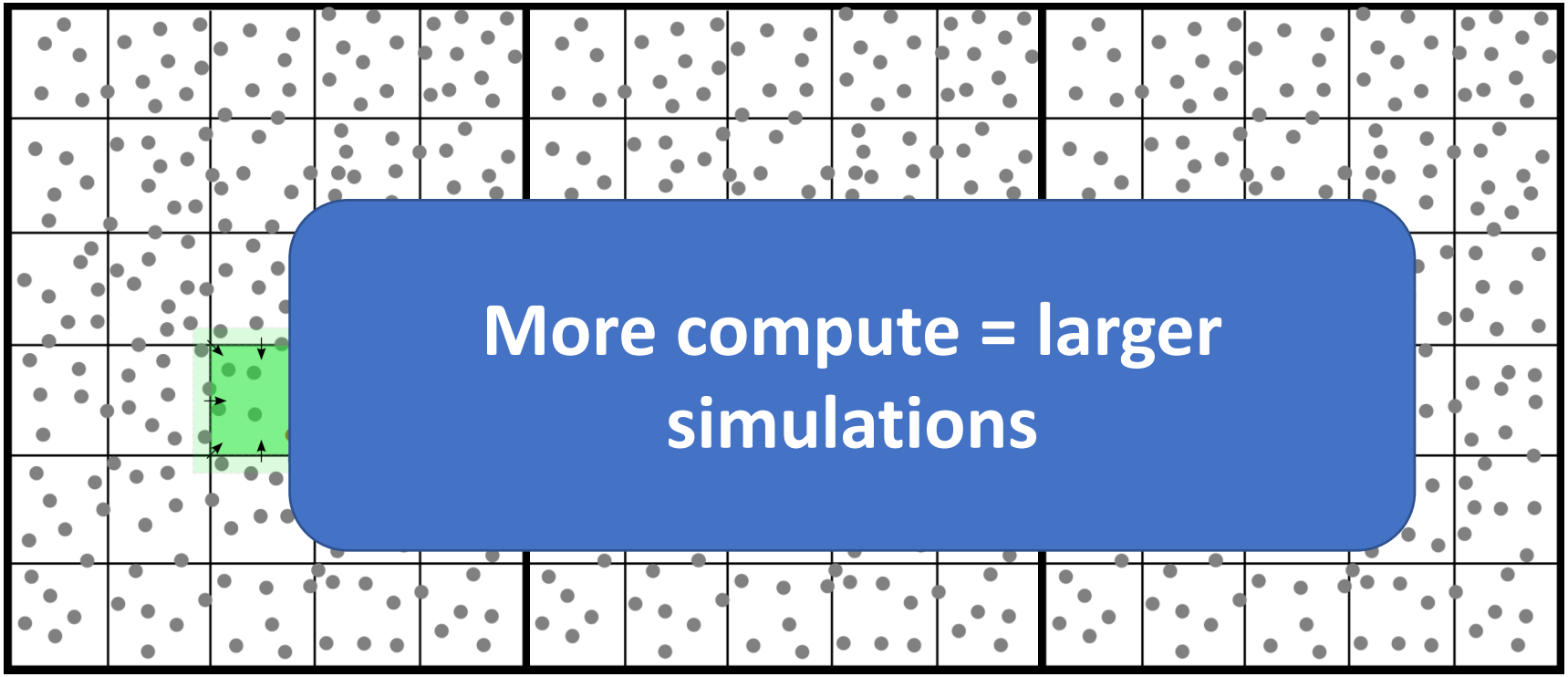
Communication required at every step



Each processor owns its domain

Scalable if
computation >> communication

MD weak-scales



Weak-scaling

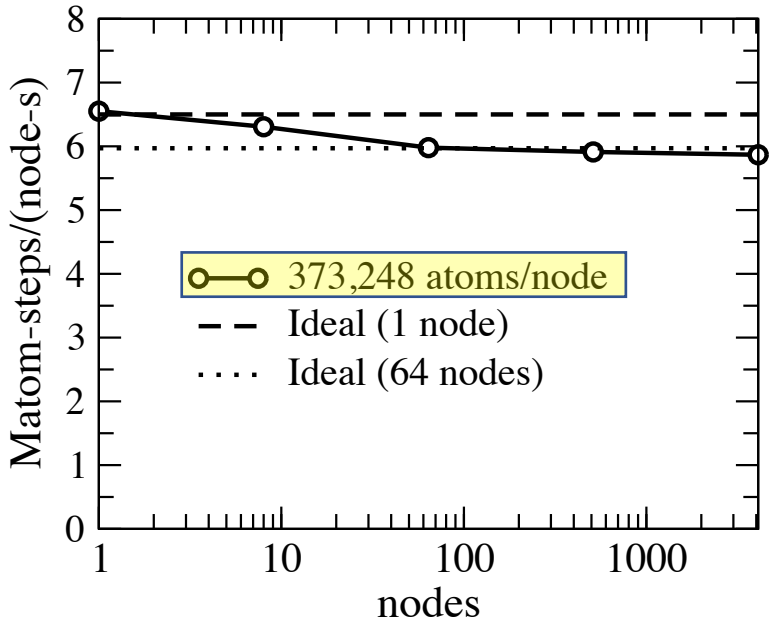
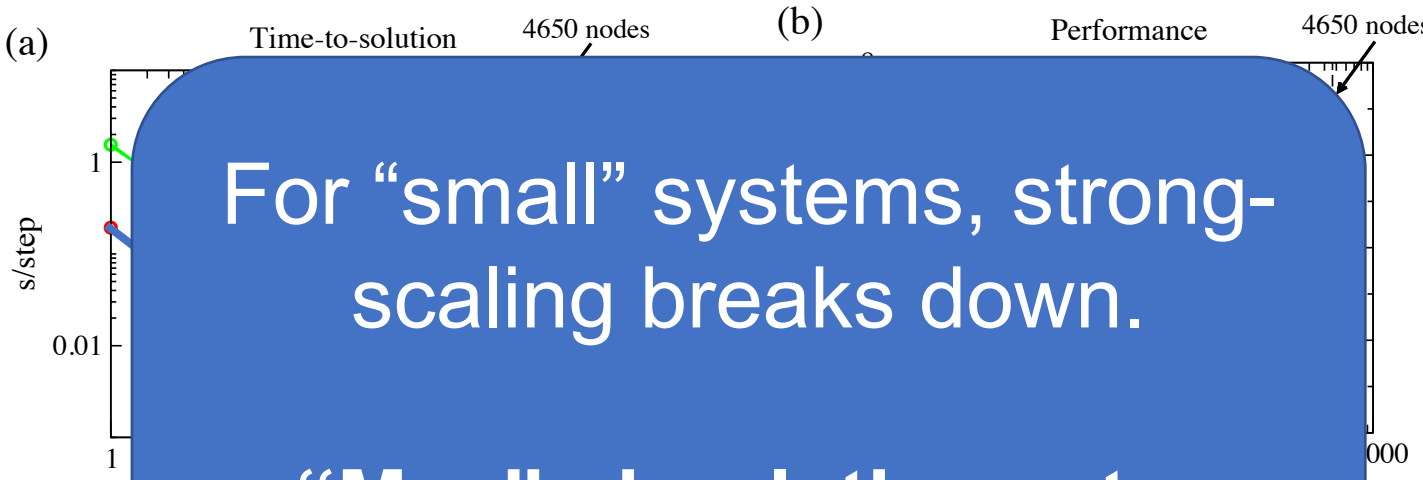


Figure 5: Weak scaling for amorphous carbon samples measured as MD performance vs node count. Sample sizes range from 373,248 atoms to 1,528,823,808 atoms and correspond to 373,248 atoms/node. Ideal scaling comparing to 1 and 64 nodes is indicated by dashed and dotted horizontal lines, respectively.

Strong-scaling



For “small” systems, strong-scaling breaks down.

“Max” simulation rate: ~10ns/day

as dashed lines. Perfect scaling in (b) would be a horizontal line (not shown).

Breakdown of strong-scaling

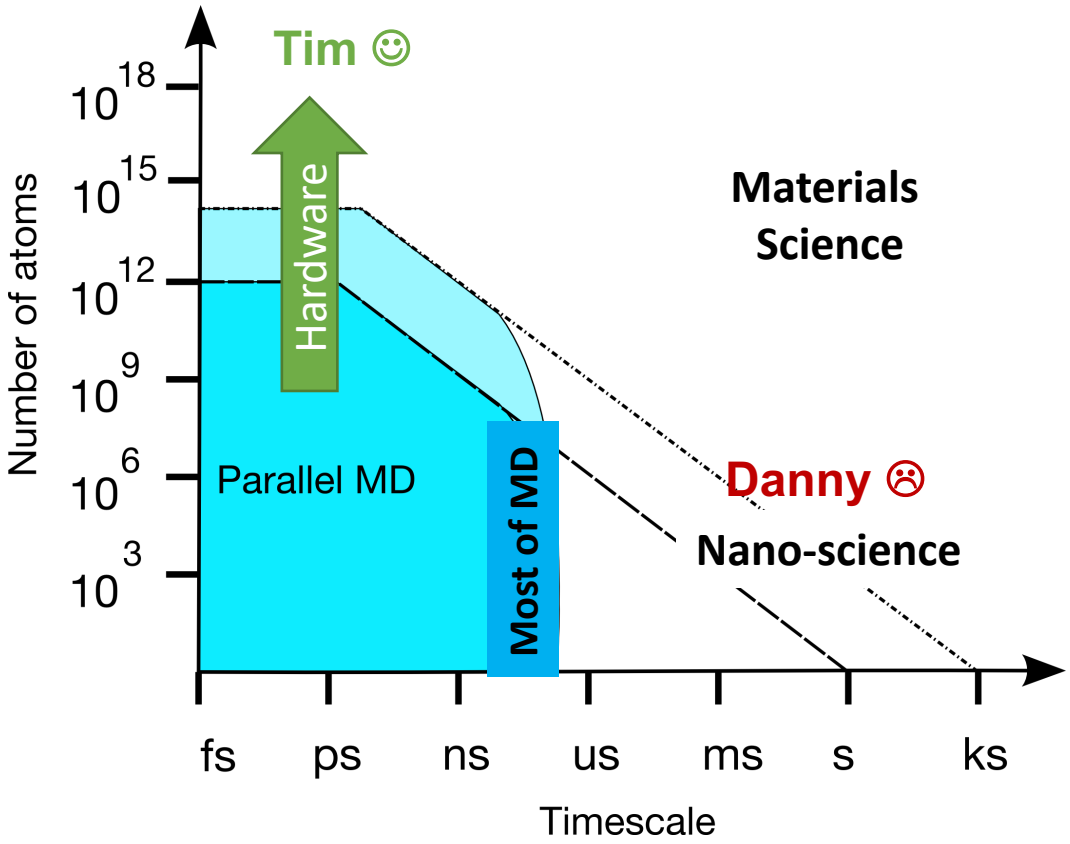
100 MILLION ATOMS	1 BILLION ATOMS	20 BILLION ATOMS

More compute ≠ longer simulations

Communication becomes the bottleneck

counts on the full machine as measured by the timers in LAMMPS. "SNAP" indicates time spent in the force computation, "MPI Comm" indicates time spent in communication, and "Other" indicates time spent in I/O, the Langevin thermostat, Verlet time integration, and other services.

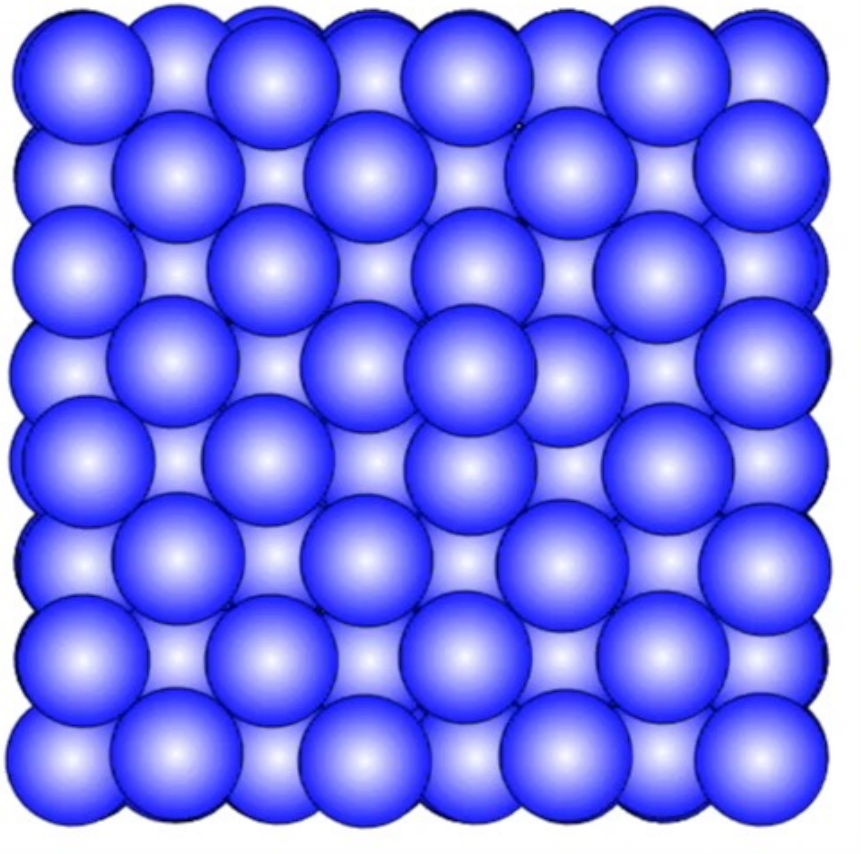
The prospect for MD at the exascale



Reaching long timescales

- How could we use exascale machines to reach long timescales?
- Parallelizing over **space** alone is not viable for small systems
- Can we parallelize over **time** instead?

Who needs long timescales anyway?



- For materials away from melting:
  - Fast vibrations/fluctuations (ps)
  - Slow conformational changes (ns-s)
- Short simulations are often **not informative** of long-time behavior
- This is bad for MD, but it is key for acceleration

Who needs long timescales anyway?

- Vibrational relaxation *within state* controlled by the curvature of V around the minimum
  - τ_{vib} ~ 1 ps
  - dt ~ 1 fs, r
    interesting

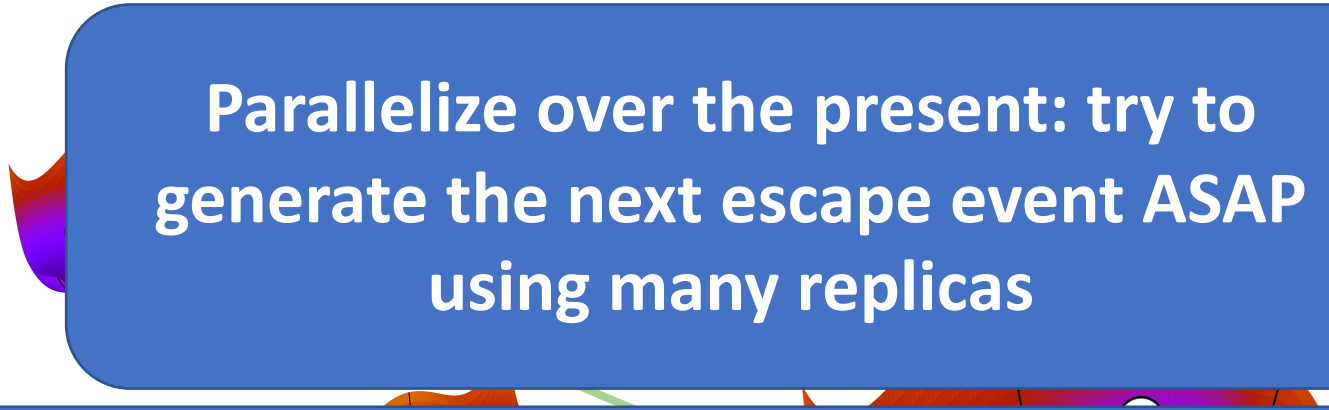
$$\tau_{esc} >> \tau_{vib} > dt \text{ whenever } \Delta E >> kT$$

- Transition *between states* requires overcoming an energy barrier ΔE
  - $\tau_{esc} \sim \tau_{vib} \exp(\Delta E/kT)$

State-to-state dynamics

Goal is to generate a single *statistically correct* state-to-state trajectory

Can we parallelize over ~~space~~ time?



Parallelize over the present: try to generate the next escape event ASAP using many replicas

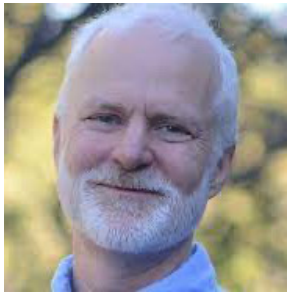
This can strong-scale scale if:

- **Simulations are independent**
- **Pieces can be spliced accurately**
- **We can use all of the pieces we generated**

[Pérez, Cubuk, Waterland, Kaxiras, Voter, JCTC 12, 18 (2016)]

Mathematicians to the rescue

- Key ideas were derived by Arthur Voter using arguments intuitive to physicists/chemists
- Turns out that “derived” has a different meaning for mathematicians…
- Working with Tony Lelièvre, Claude Le Bris, Mitch Luskin, we endeavored to clean things up. First meeting at IPAM about 13 years ago…
- Lead to a much more sophisticated understanding of the generality of the methods

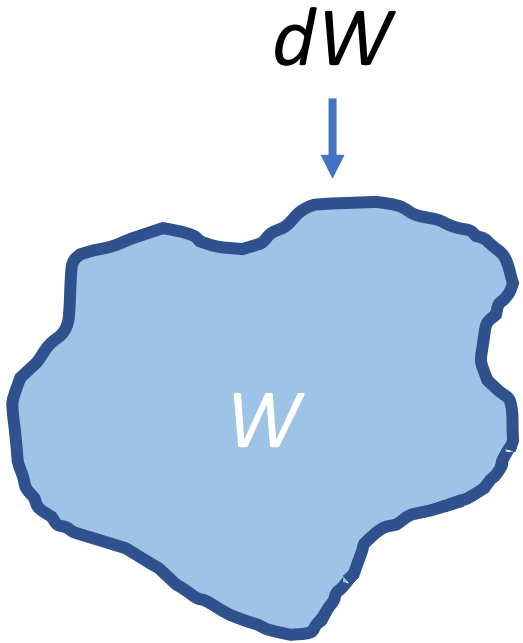



State-to-state dynamics

Need to capture **transition statistics**:

- Distribution of first-escape times from *W*
- Distribution of first-escape points on *dW*

Key Concept: Quasi-stationary Distribution (QSD)

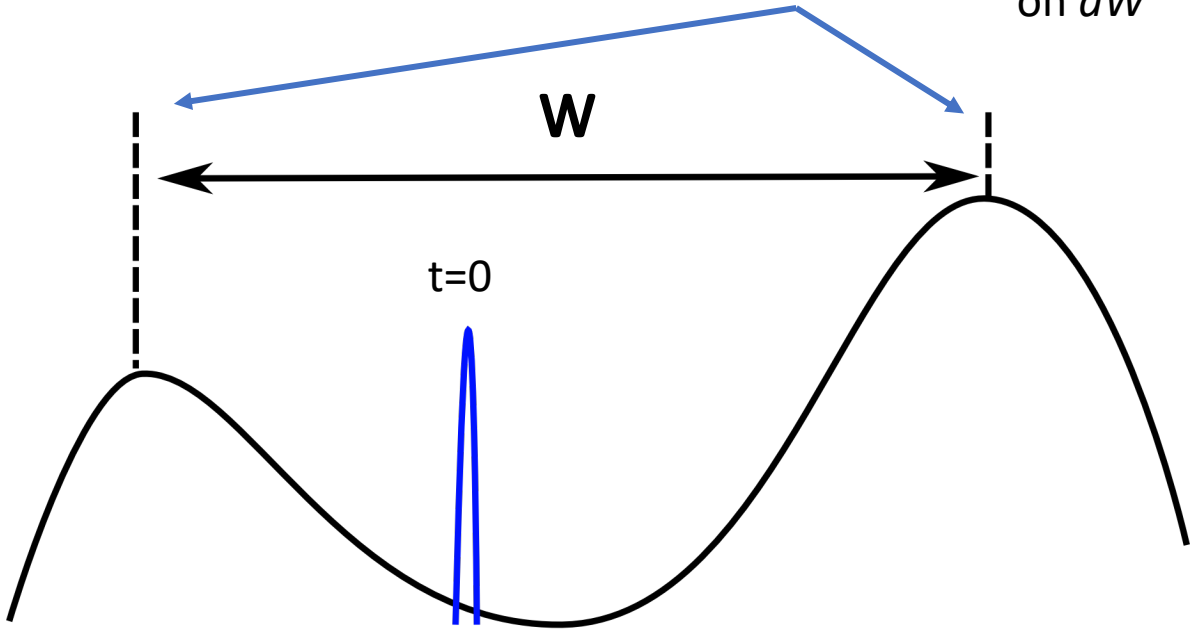


$$\nu(A) = \frac{\int_W \mathbb{P}(X_t^x \in A,\, t < T_W^x)\, d\nu}{\int_W \mathbb{P}(t < T_W^x)\, d\nu}.$$

If X_0 is distributed according to *QSD*, then, conditionally on not having left W up to time t, X_t is still distributed according to *QSD*

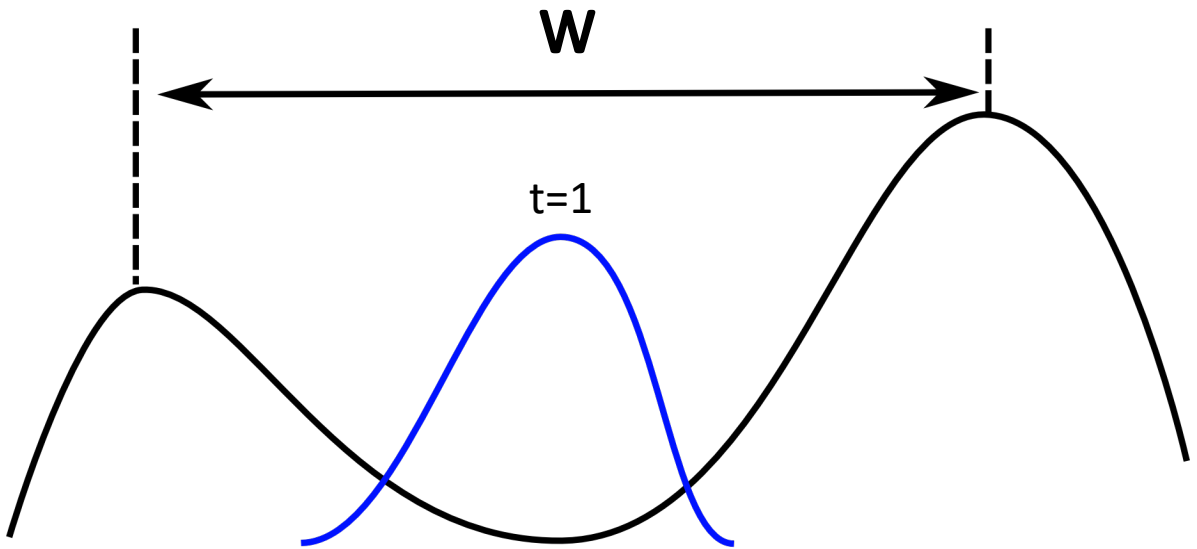
Quasi-stationary distribution

Impose absorbing boundaries on *dW*



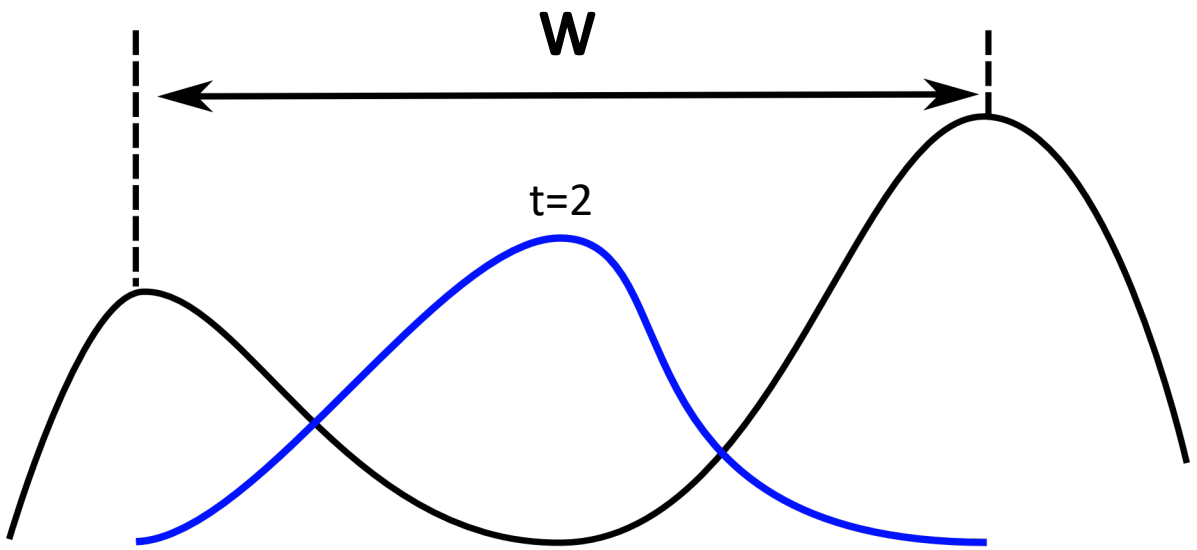
Consider an ensemble of trajectories initialized somewhere in state W

Quasi-stationary distribution



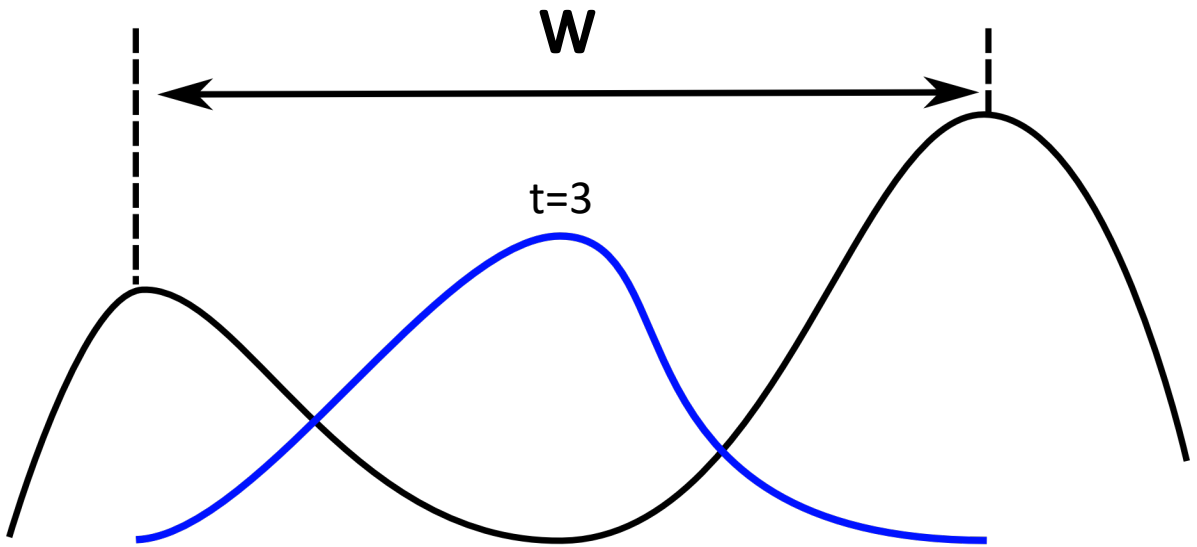
Evolve that ensemble in time, removing any trajectory that escapes

Quasi-stationary distribution



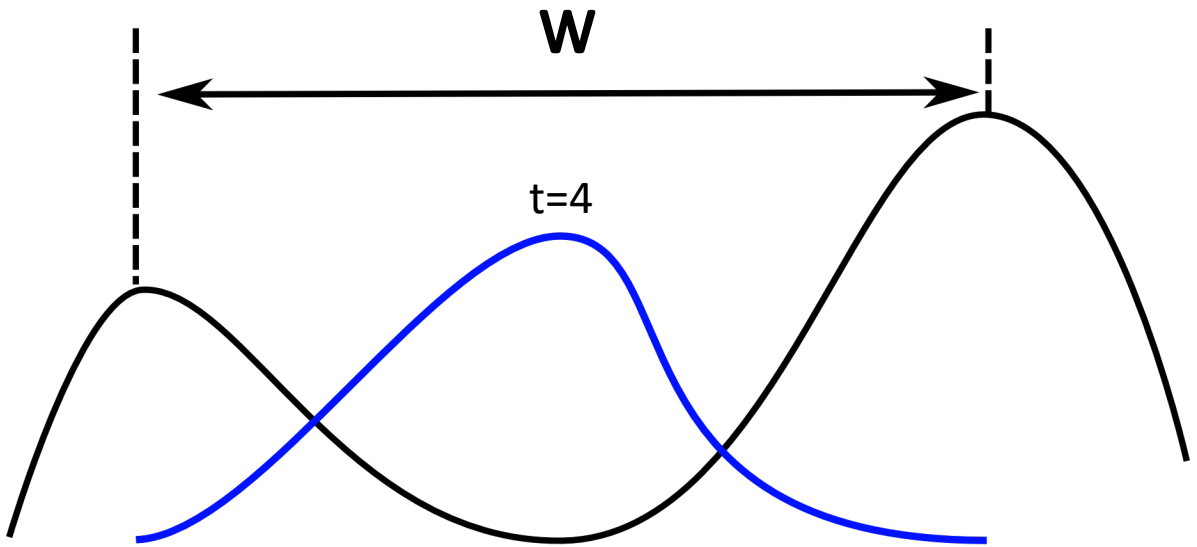
Look at the distribution of whomever is left

Quasi-stationary distribution



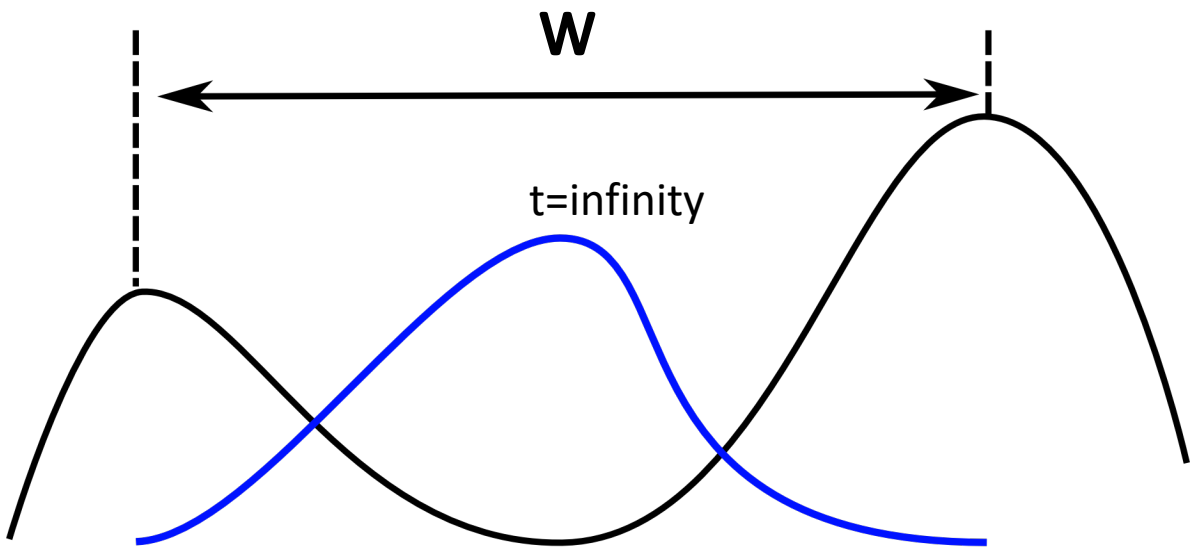
That distribution eventually converges…

Quasi-stationary distribution



And no longer varies with time…

Quasi-stationary distribution



This limiting distribution is the QSD of state W

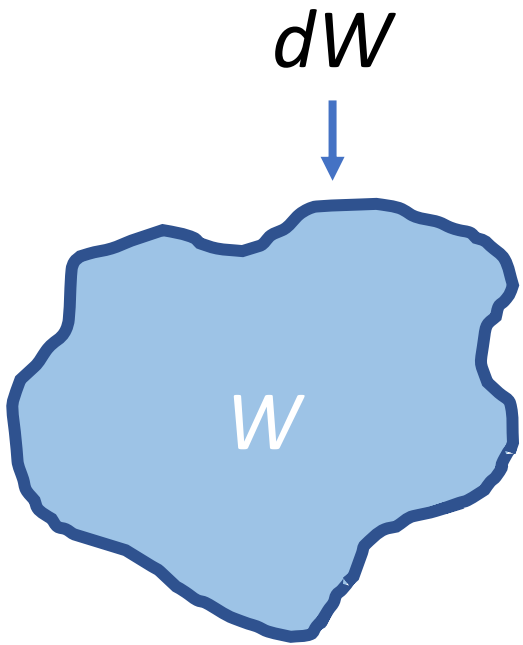
QSD for Langevin dynamics

In the following:

- Overdamped Langevin dynamics
- Absorbing boundary conditions on *dW*
- Generator has eigenvalues $0 > -\lambda_1 > -\lambda_2 \geq -\lambda_3$ ...
- **QSD is eigenfunction $u_1(X)$ of generator corresponding to λ_1**

Most of the following also applies to other dynamics, if:

- QSD exists
- QSD is unique
- Convergence to the QSD is fast



QSD for Langevin dynamics

$$\frac{\partial \rho}{\partial t} = L\rho \text{ on } W$$
$$\rho = 0 \text{ on } \partial W$$

With $L = -\nabla V \cdot \nabla + \beta^{-1}\Delta$

Then:

$$\rho(X,t) = \sum_k e^{-\lambda_k t}\, c_k^0 u_k(X)$$

For $t > (\lambda_2 - \lambda_1)^{-1}$ and conditional on not having escaped,

$$\hat{\rho}(X,t) \cong u_1(X) + O(e^{-(\lambda_2-\lambda_1)t})$$

Properties of the QSD

Rate of memory loss

- The QSD of W is **unique**
- Convergence to the QSD is **exponential** with rate $(\lambda_2 - \lambda_1)$

From the QSD:

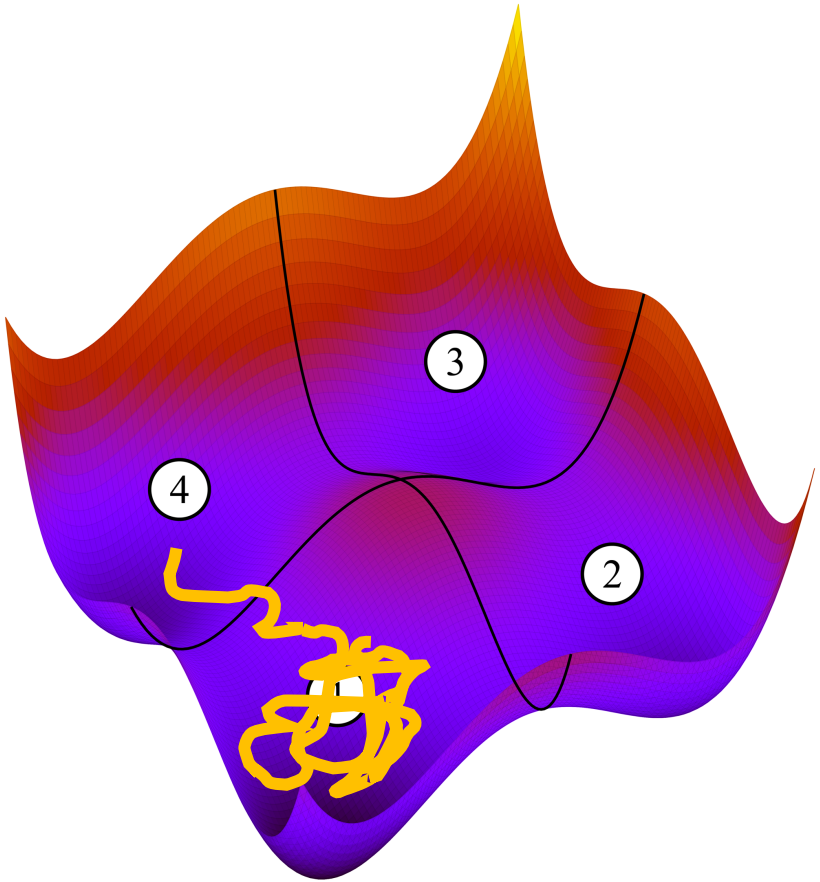
- First escape time is random and exponentially distributed with rate λ_1
- First escape point is random and uncorrelated with escape time

This is true for any state definition!

Does not depend on history before reaching the QSD

Overdamped Langevin: [Le Bris, Lelievre, Luskin, and DP, MCMA 18, 119 (2012)]

Langevin: [Lelievre, Ramil, Reygner, arXiv:2101.11999]



After only a short time in the state, the next escape time/location distribution is a complex function of the entry point

After spending $t_c > (\lambda_2 - \lambda_1)^{-1}$ in W, the next escape from W becomes *Markovian

All trajectories that spent $t_c > (\lambda_1 - \lambda_2)^{-1}$ in W are statistically equivalent with respect to how and when they will leave W*

* Up to an exponentially small error in t_c

Tra

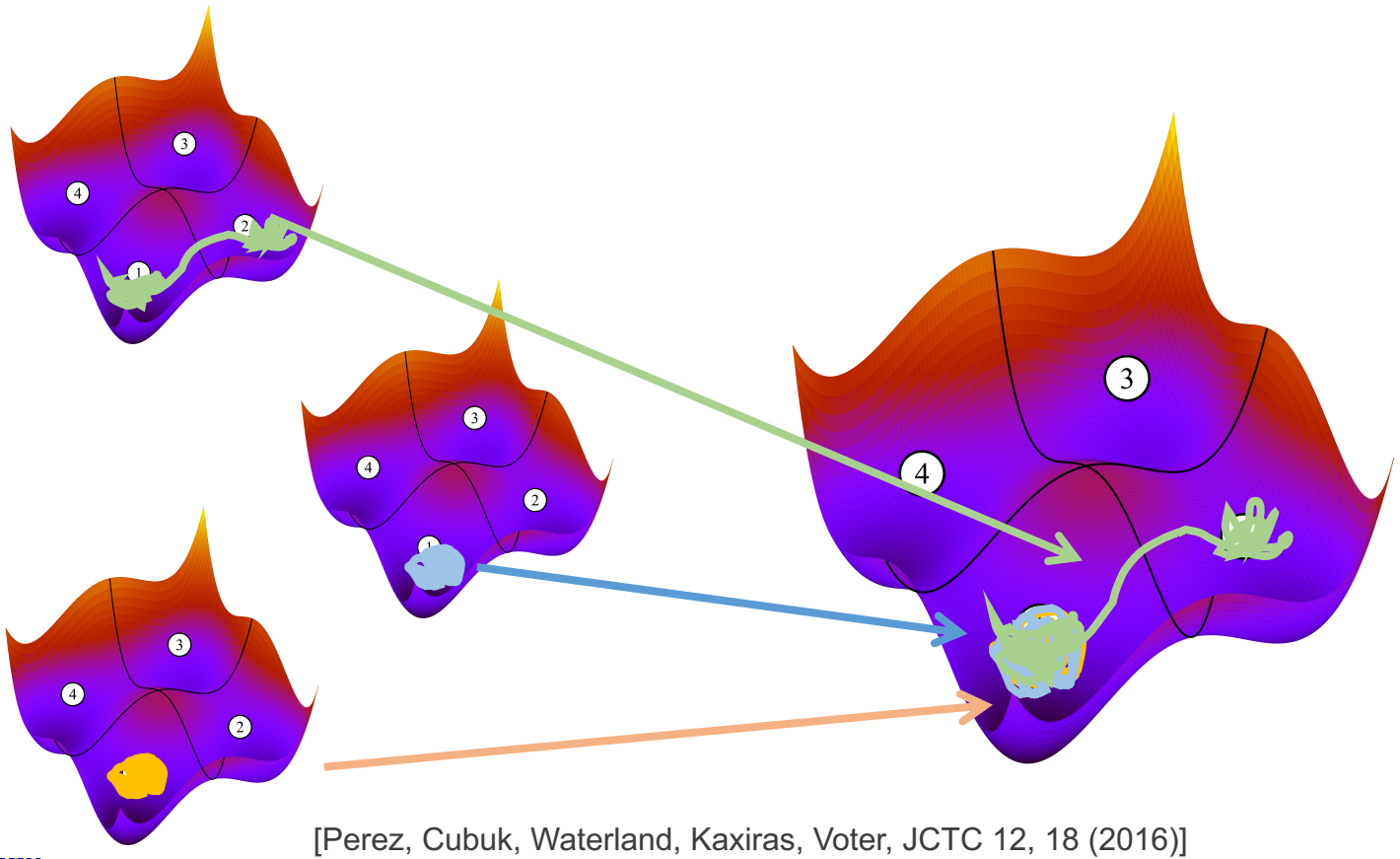
D

A valid state-to-state trajectory can be assembled by splicing *independent* segments end-to-end*

Exactly what we need for strong-scaling!

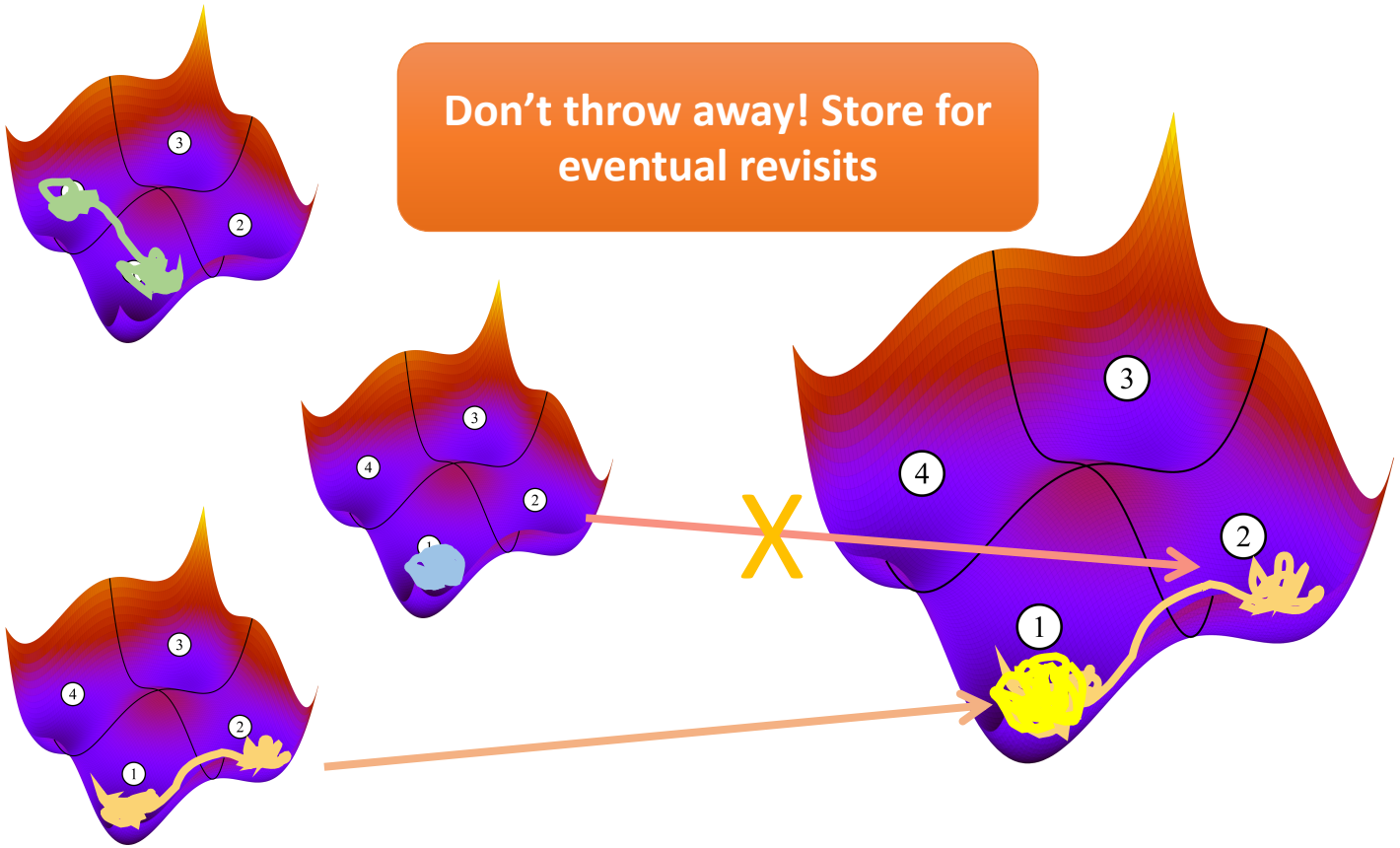
* Up to an exponentially small error in t_c

Parallel Trajectory Splicing (ParSplice)



[Perez, Cubuk, Waterland, Kaxiras, Voter, JCTC 12, 18 (2016)]
[Aristoff, SIAM/ASA Journal on Uncertainty Quantification 7, no. 2 (2019): 685-719]

Parallelizing over the past with bookkeeping



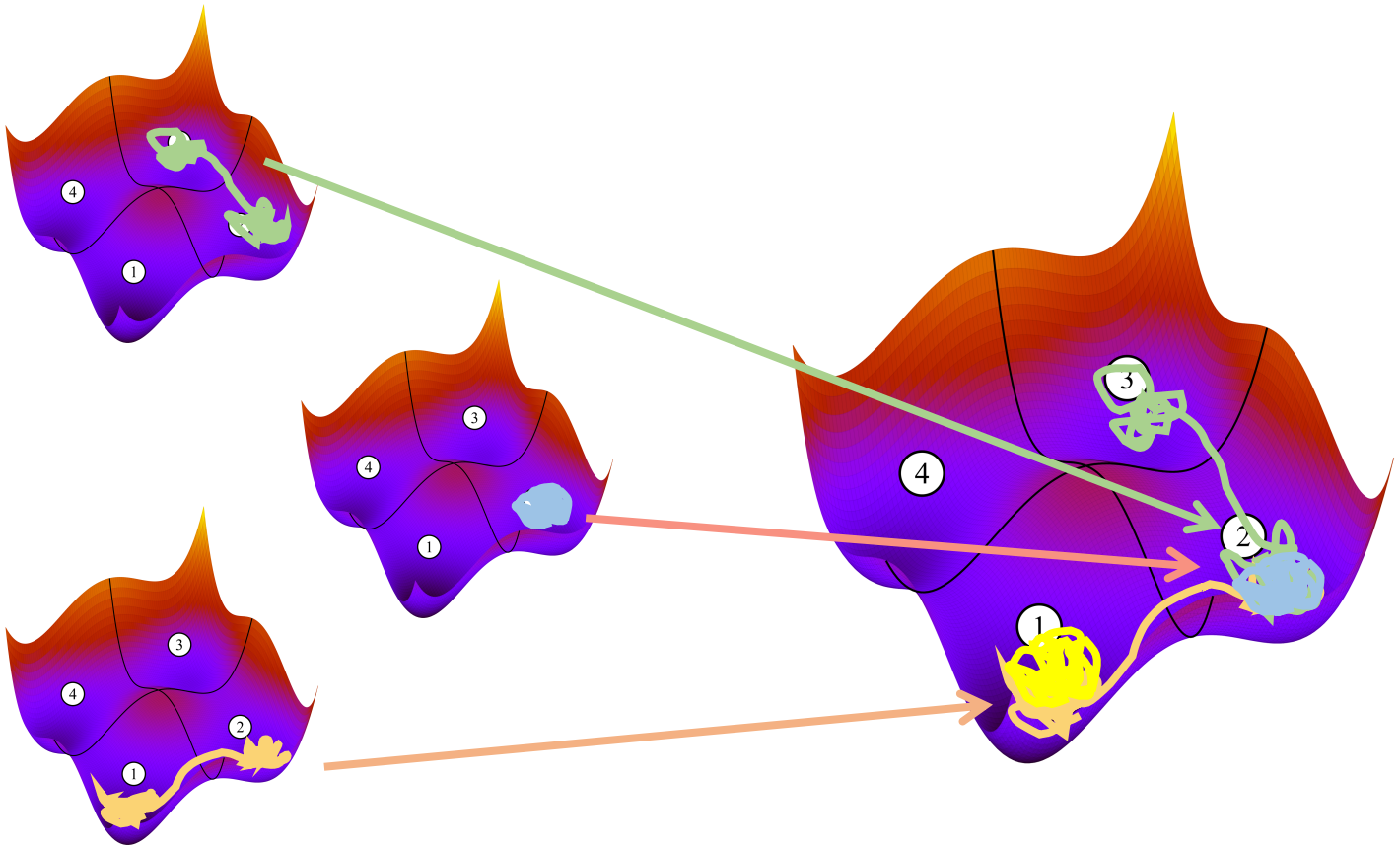
Super-basins

Parallelize over the past: store work done but not used for (potential) future use.

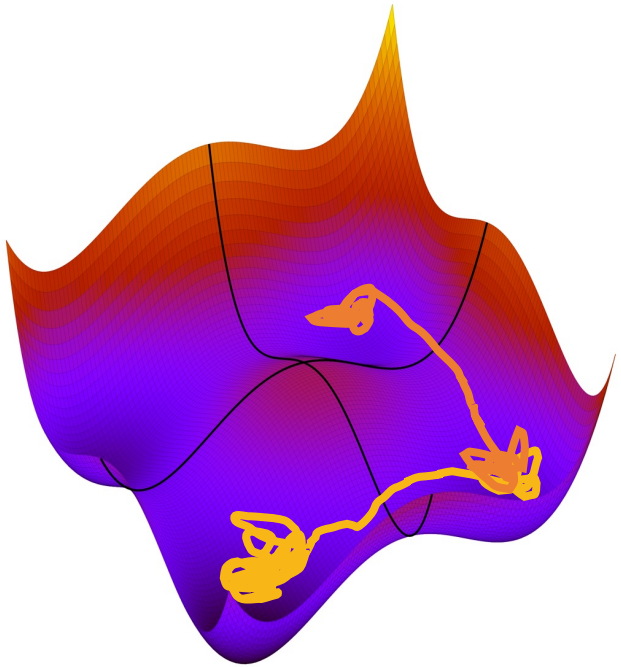
Precomputation/caching is a common strategy

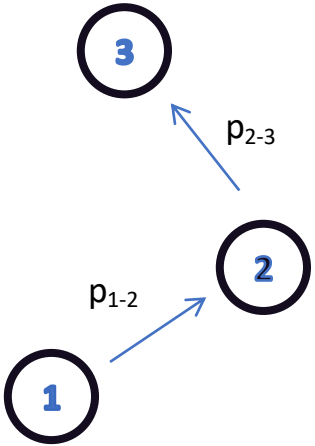
ly
rials

Parallelize over the future with speculation



Statistical oracle





Statistical oracle

We use this model to speculate where the trajectory will be in the future

ot

Parallelize over the future: allocate work based on where you think the system will be in the future

Speculation is a common strategy at the instruction level (branch prediction), but not so much the task level

More on this in WS 1

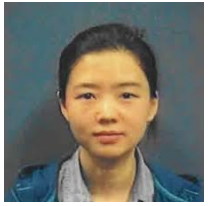
See A. Garmon,

See A. Garmon, V. Ramakrishnaiah, DF, arXiv:2010.11792, for use of model for resource allocation

Shape Fluctuations in Nanoparticles

- Metallic nanoparticles (150-300 atoms)
- Between 3,600 and 36,000 cores

- **Long simulations:** up to 4 ms
- **Many transitions:** up to ~100M per run
- **Many states:** up to ~1M per run



Rao Huang
(Xiamen U.)

Huang, Lo, Wen, Voter, Perez, JCP 147, 152717 (2017)
Perez, Huang, Voter, JMR 33, 813 (2018)
Huang, Wen, Voter, Perez, Phys. Rev. Mat. 2, 126002 (2018)

Element	Number of Atoms	T (K)	Trajectory Length (ps)	Number of Transitions	Number of States	Description
Pt	146	900	70,257,528	162,965	6,246	fcc ⇒ deca ⇒ ico
	170	800	672,396,434	1,937,031	147,377	fcc ⇔ 5-fold caps ⇒ ico
		900	20,373,095	240,306	117,680	
	190	800	1,350,168,728	6,630,131	303,572	----------
		900	348,662,895	688,027	93,346	fcc ⇒ ico
	231	900	1,986,709,692	4,395,285	252,153	----------
		1000	92,171,602	955,401	42,383	
		1100	24,608,419	914,005	110,290	
Cu	146	550	301,832,137	3,942,180	237,293	fcc ⇒ ico
	170	500	4,156,073,707	6,160,286	240,594	----------
		550	23,712,165	656,202	241,491	fcc ⇔ 5-fold caps ⇒ ico
		600	21,690,608	1,039,065	144,713	fcc ⇒ deca ⇒ ico deca⇒ fcc ⇒ ico
	190	500	489,113,720	93,863,998	368,356	----------
		600	91,701,072	9,863,950	847,016	
	231	500	438,302,547	49,409	12,817	
		550	66,578,597	4,623,717	262,785	
		600	85,056,822	184,737	169,217	
		700	832,190	237,840	89,356	
Au	146	600	237,233,817	22,910,983	119,489	fcc ⇒ ico
	190	600	521,506,615	10,198,278	85,875	fcc ⇔ 5-fold caps
	231	800	774,813,889	795,678	159,743	fcc ⇒ 5-fold caps ⇒helical
Ag	146	500	122,897,307	2,558,937	71,357	----------
		550	21,613,546	1,988,646	136,297	fcc ⇔ off-centered 5-fold axis
	170	500	841,036,559	1,529,663	258,281	----------
		600	128,965,726	3,961,585	616,430	
	190	400	1,651,496,973	2,416,400	60,802	
		500	109,165,848	1,414,790	154,083	
		600	30,620,753	1,091,307	147,863	
	231	500	20,445,451	946,623	92,818	

Benchmark results: An Easy Case

Rare events

T=300K, LANL Grizzly, 4h runs

N_{cores}	Trajectory length (ps)	Generated segment time (ps)	#Transitions	#States	$\langle t_{trans}/N_{tc} \rangle$	<R>	Simulation rate (μs/hour)
9,000	556,093,988	556,539,980	4,614	28	13.[illegible]9	166	139
18,000	1,315,941,923	1,346,516,503	24,610	64	2.[illegible]7	384	333
27,000	2,209,432,238	2,214,868,608	13,479	47	4.[illegible]5	294	552
36,000	2,291,027,808	2,318,254,470	50,258	60	1.26	909	592

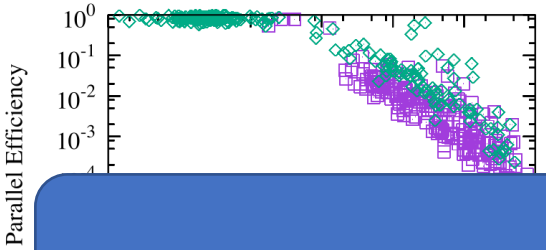
99% of generated segments were spliced

Peak simulation rate: 10 μs/min, 10 ms/day

Benchmark results: Hard Cases

~4x from speculation

Most of performance from revisits



	Trajectory length (ps)	Generated segment time (ps)	#Transitions	#States	$\langle t_{trans}/M\ t_c \rangle$	<R>	Simulation rate (µs/hour)
				15	21.25	149	85
400	267,608,621	305,631,041	21,629,711	4,785	0.0017	545	69
				24,161	0.00029	3846	48
				250,867	0.00028	135	17
				36,513	0.00033	20	0.45
900	169,943	10,673,302	64,208	11,577	0.00030	6	0.043

Multiple levels of parallelism contribute in hard cases

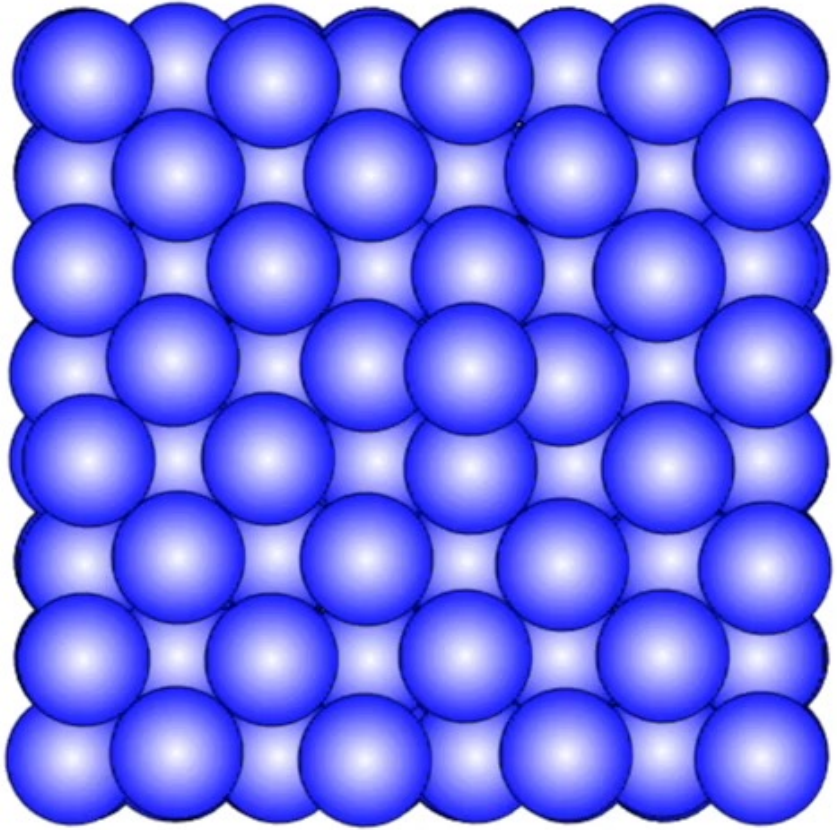
Reduces to MD if dynamics is fast and new/unpredictable things happen all the time

75% of generated segments were spliced
2700x speedup over MD

Very fast events: need only a few segments to escape

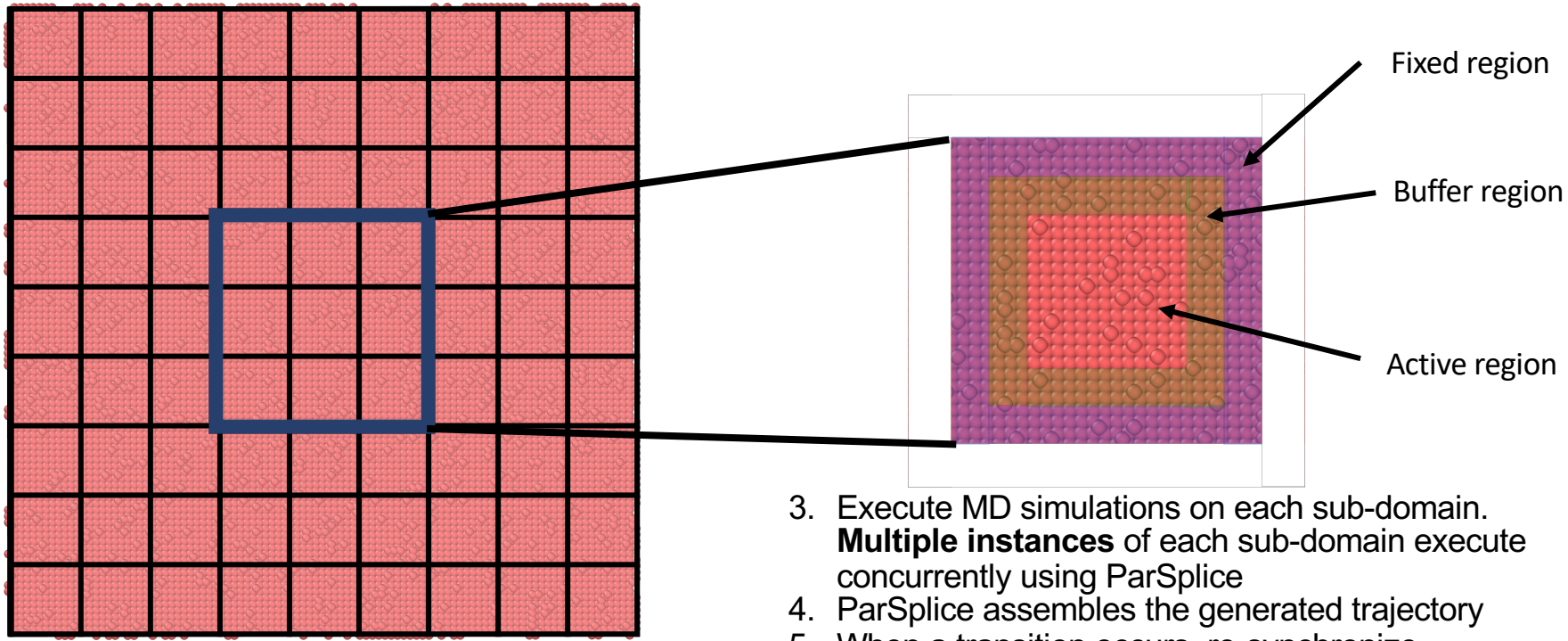
Parallelizing over space

- The efficiency of ParSplice is limited by the **global** rate at which events occur: **the rarer the better**
- This limits performance on large systems where the **exit rate scales with N**
- However, most transitions are spatially **localized.** Can we use this to make ParSplice sensitive only to the **local** event rate?



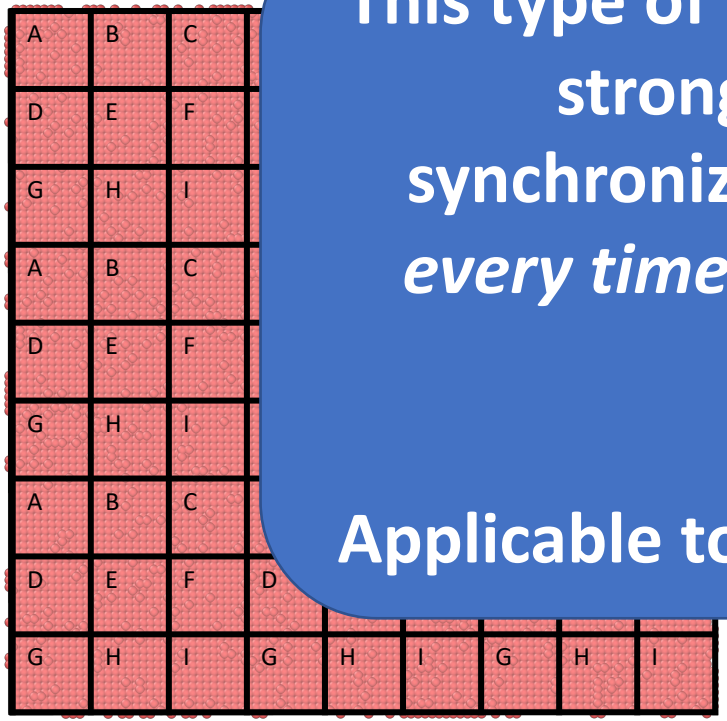
Spatial Parallelization Strategy

1. Divide the simulation cell into a grid of sub-domains.
2. Extract and prepare the sub-domains



3. Execute MD simulations on each sub-domain. **Multiple instances** of each sub-domain execute concurrently using ParSplice
4. ParSplice assembles the generated trajectory
5. When a transition occurs, re-synchronize neighboring sub-domains

Domain synchronization



This type of domain decomposition is strong-scalable because synchronization is *not* required at *every timesteps*, but only at *every (rare) event.*

Applicable to many different methods

same

e for a

ze

vent

[Shim and Amar, Phys. Rev. B 71, 115436 (2005)]

How to run this at scale? ParSplice as a workflow

- Short tasks (~sec)
- Limited scalability per task (~1 GPU - a few nodes)
- Inter-task dependencies
- *Just-in-time* task identification
- Large computing resources

- **Many, short, tightly-coupled tasks**

- Scheduling with queuing system
- File-based communication
- Synchronous communication

- **Internal task management**
- **File-less API-based engines**
- **Asynchronous communication and I/O**

High-level model

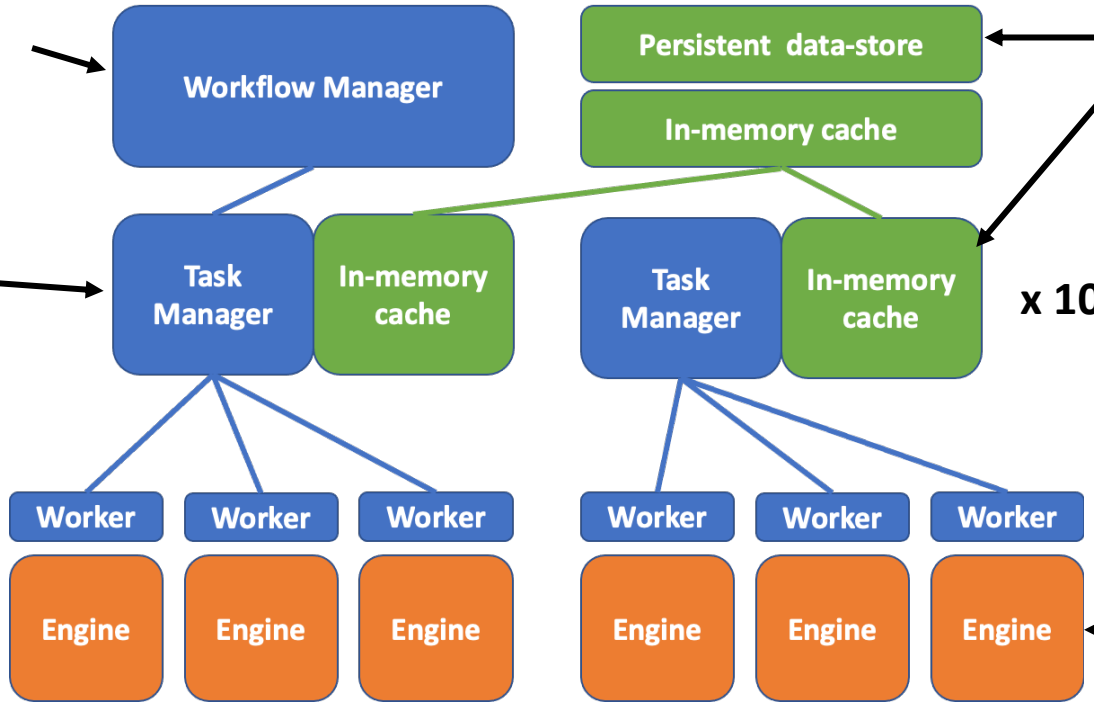
- Top-down scheduling of complex workflows on large heterogeneous hardware is hard
- EXAALT model: **let hardware availability drive execution**
- EXAALT is a **PULL** model: the framework **requests tasks** from the workflow to maintain high hardware usage
- **The workflow should be able to identify new tasks at any time**

Under the hood of EXAALT

Task management
Data management
Task execution

Generate tasks, process results

Manage task execution

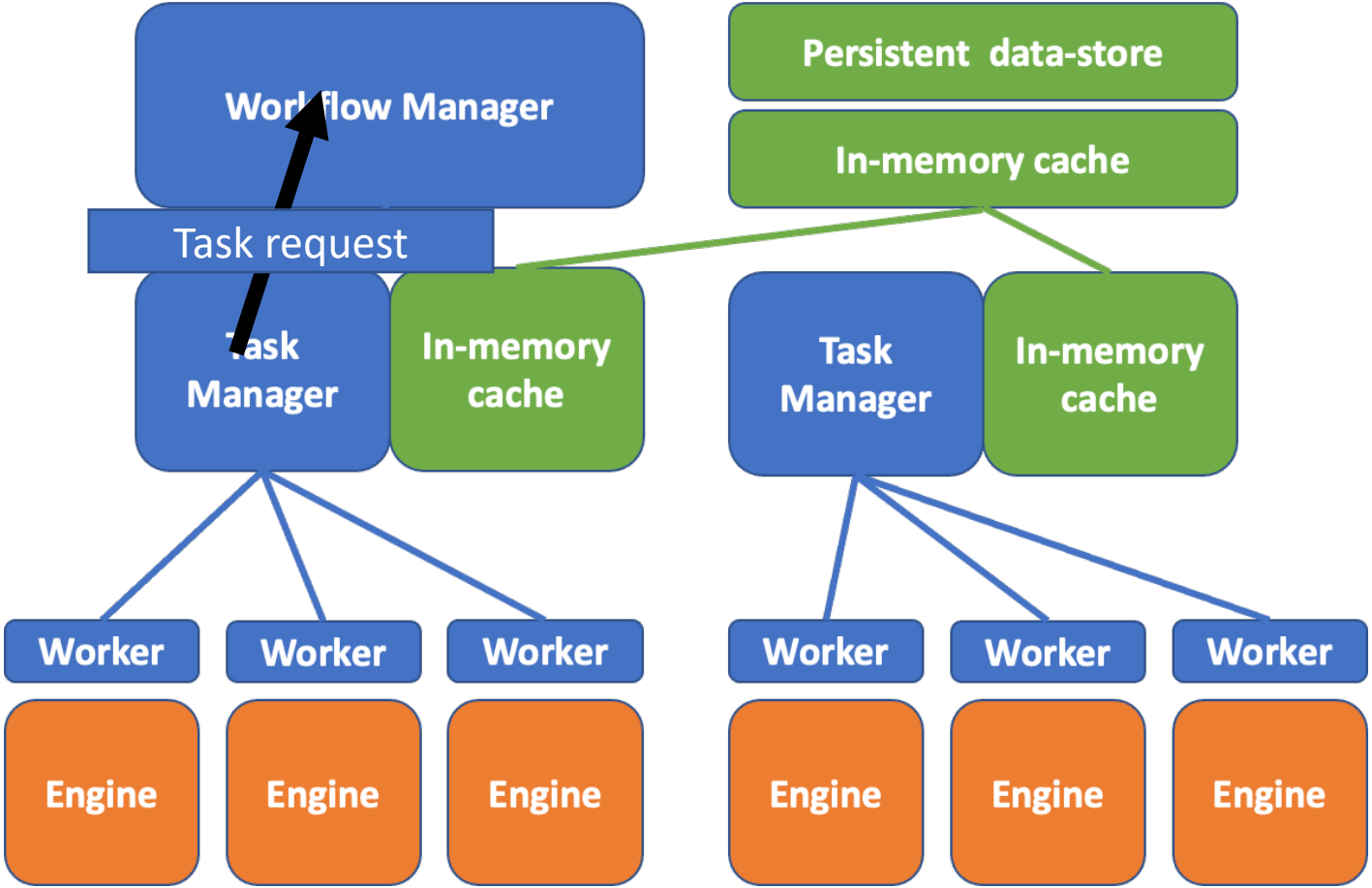


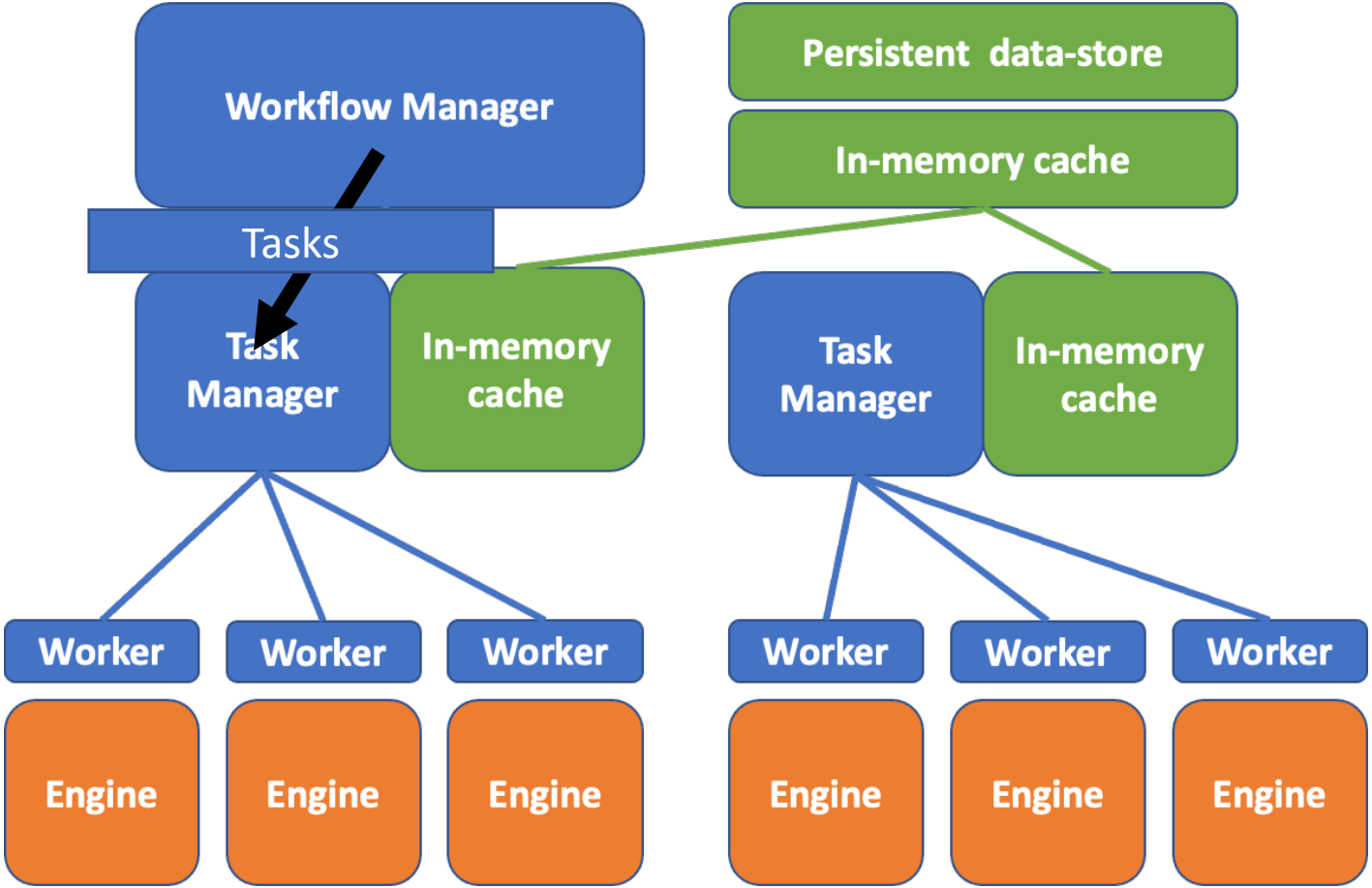
Store/move data

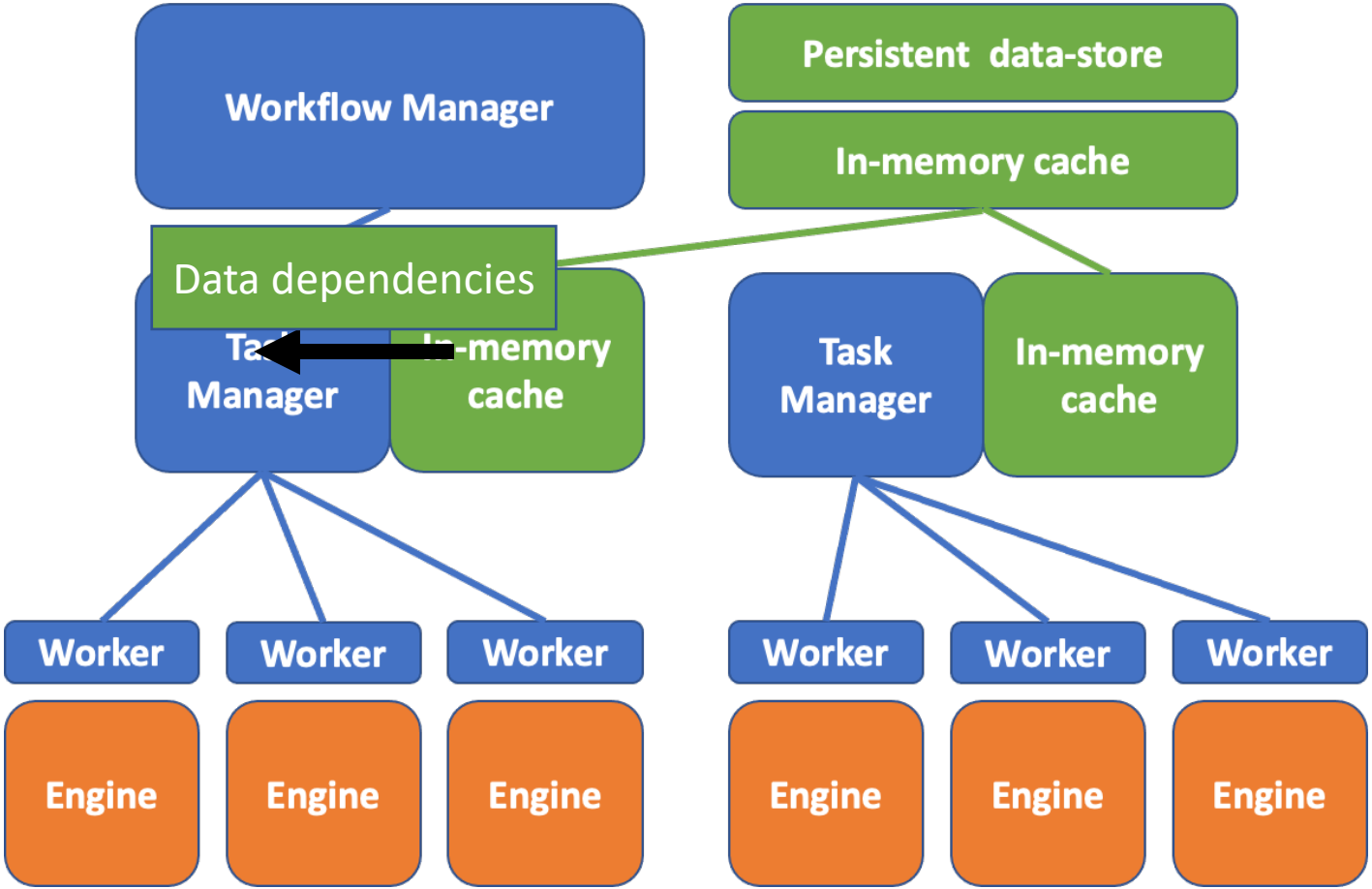
x $10 – 10^2$ managers

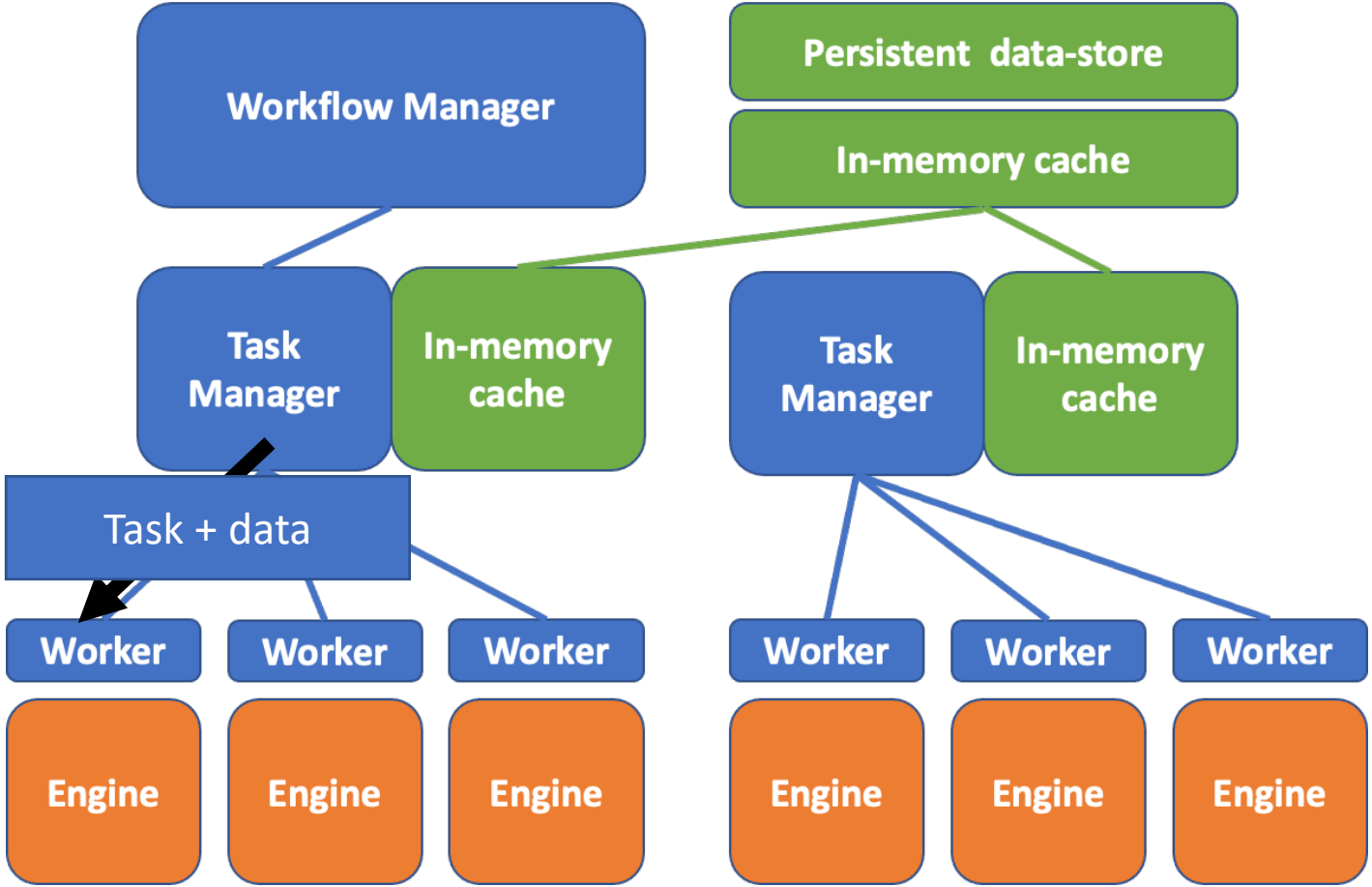
Execute tasks

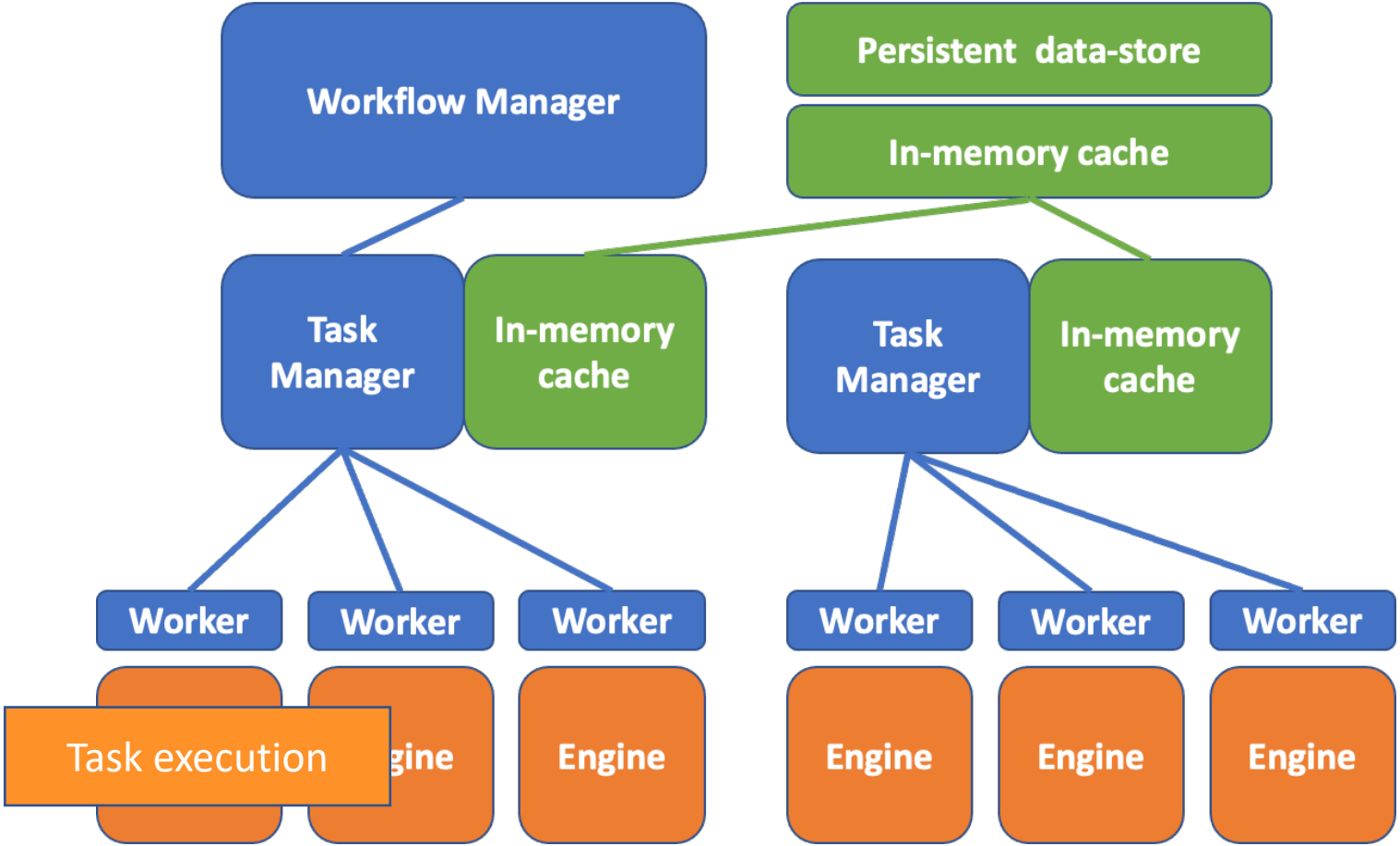
x $10^3 – 10^6$ workers

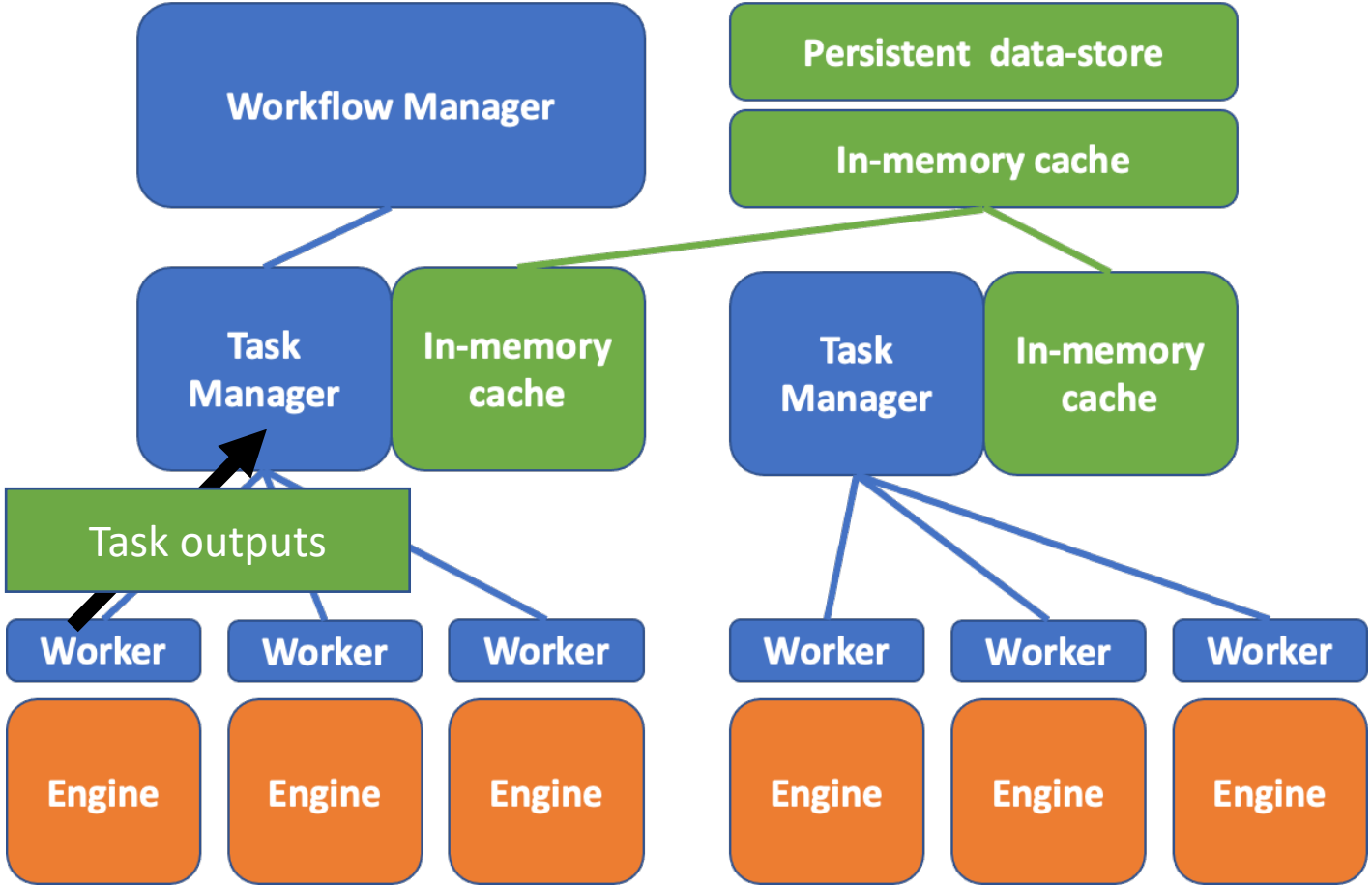

Workflow Manager
Persistent data-store
In-memory cache
Task request
Task Manager
In-memory cache
Task Manager
In-memory cache
Worker
Worker
Worker
Worker
Worker
Worker
Engine
Engine
Engine
Engine
Engine
Engine

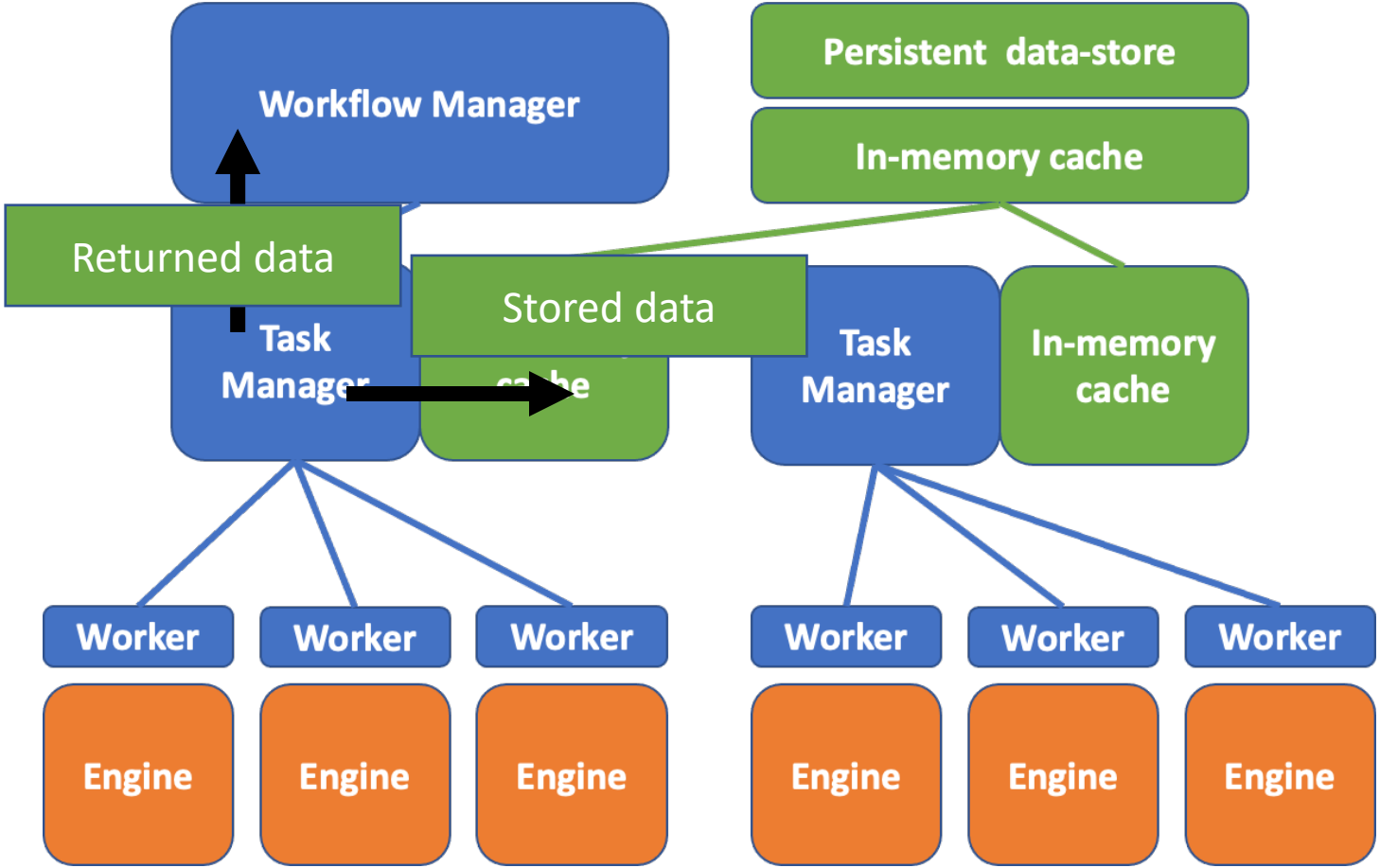

Workflow Manager
Tasks
Persistent data-store
In-memory cache
Task Manager
In-memory cache
Task Manager
In-memory cache
Worker
Worker
Worker
Worker
Worker
Worker
Engine
Engine
Engine
Engine
Engine
Engine


Workflow Manager
Persistent data-store
In-memory cache
Data dependencies
Task Manager
In-memory cache
Task Manager
In-memory cache
Worker
Worker
Worker
Worker
Worker
Worker
Engine
Engine
Engine
Engine
Engine
Engine


Workflow Manager
Persistent data-store
In-memory cache
Task Manager
In-memory cache
Task Manager
In-memory cache
Task + data
Worker
Worker
Worker
Worker
Worker
Worker
Engine
Engine
Engine
Engine
Engine
Engine


Workflow Manager
Persistent data-store
In-memory cache
Task Manager
In-memory cache
Task Manager
In-memory cache
Worker
Worker
Worker
Worker
Worker
Worker
Engine
Engine
Engine
Engine
Engine
Engine
Task execution


Workflow Manager
Persistent data-store
In-memory cache
Task Manager
In-memory cache
Task Manager
In-memory cache
Task outputs
Worker
Worker
Worker
Worker
Worker
Worker
Engine
Engine
Engine
Engine
Engine
Engine

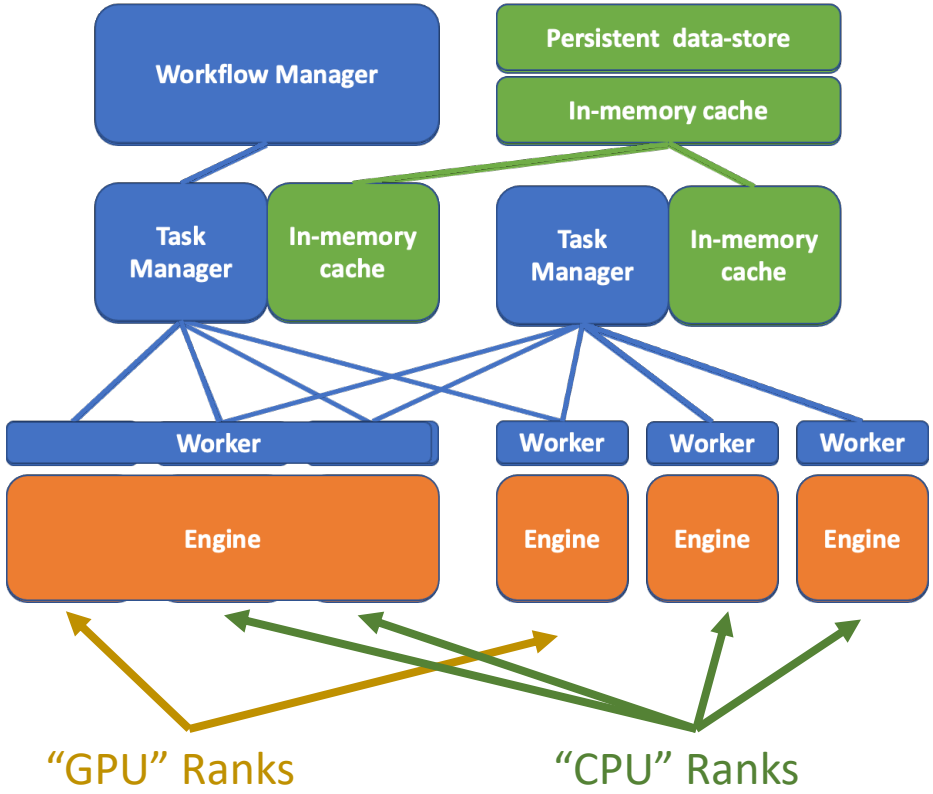

Workflow Manager
Persistent data-store
In-memory cache
Returned data
Stored data
Task Manager
Task Manager
In-memory cache
Worker
Worker
Worker
Worker
Worker
Worker
Engine
Engine
Engine
Engine
Engine
Engine

Design goals/choices

- **Pull model:** the hardware is telling you what it wants
- **No worker should ever be idle:**
  - Communication/data motion should occur in the background
- **Everything is asynchronous**/non-blocking:
  - Communication between WM and TM
  - Access to datastore
- Flat consumer-producer model not scalable: use TMs as middle-men
  - Maintain local task queues and fulfills data dependencies
  - TM pre-emptively **requests more tasks *before*** running out
  - Aggregates small messages into larger ones
  - TMs can be hardware-specific

Support for heterogeneous hardware

- MPI ranks assigned to each TM at launch
- Can tie specific hardware to specific TM: “GPU” TM and “CPU” TM and route tasks accordingly
- Can **dynamically** adjust granularity of workers under each TM.



EXAALT

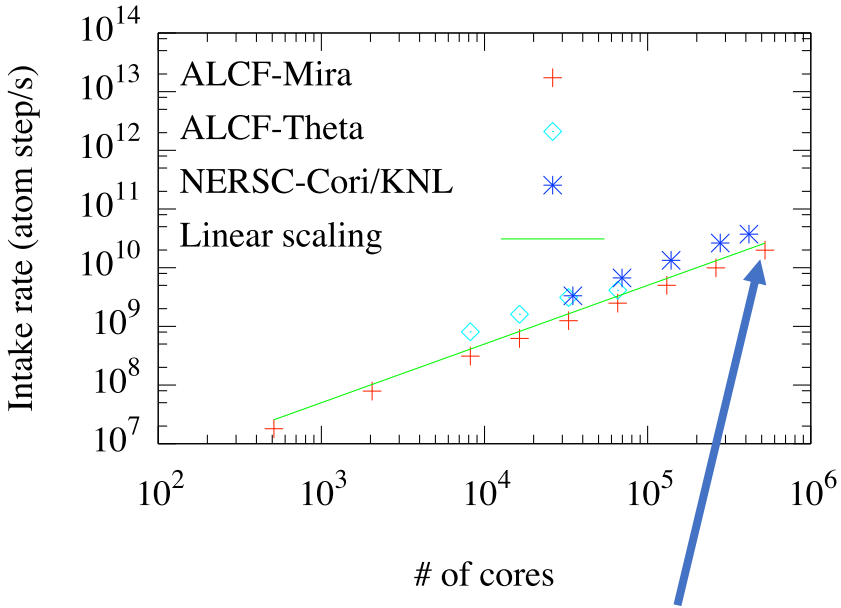
Pros:

- Very scalable!
- Simple API
- Python bindings
- Growing engine ecosystem
  - Native python code
  - MD: LAMMPS
  - QM: NWChem, Quantum Expresso, DFT-FE, LATTE
  - KMC: SPARRKS
  - DDD: ModeLIB (DDD) [in development]
  - Support for MolSSI/MDI plugin model

Cons:

- Not designed for all workloads
- Early stage of development

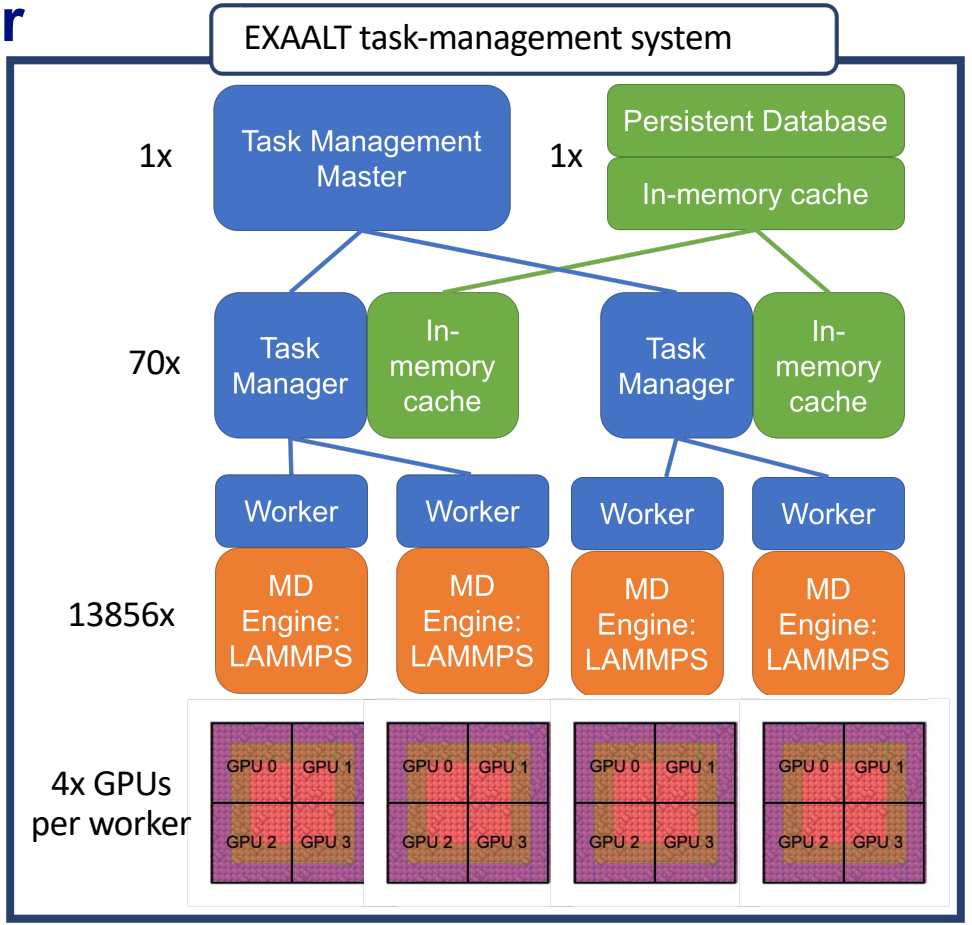
MD intake rate EAM potential
Peak: 6×10^{10} atom-step/s



~50,000 tasks/sec

Mapping ParSplice unto Frontier

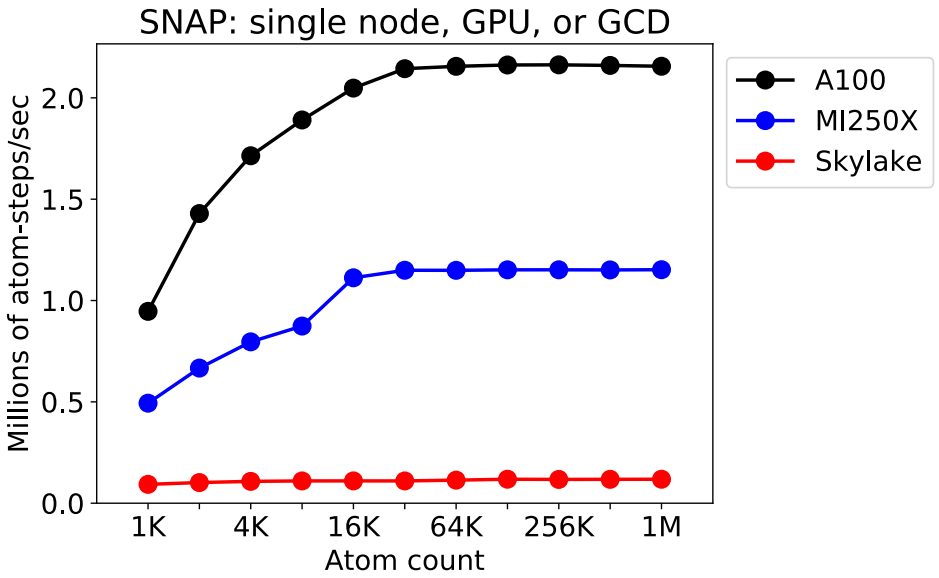
- Simulation executed using EXAALT on 7000 Frontier nodes (75% of machine)
- ~1% of resources for management
- ~99% of resources to simulation
- Infrastructure re-assigns MD tasks to workers every ~7 seconds
- 81 sub-domains
- ~170 instances of each sub-domain execute concurrently
- 4 GPU dies for every instance



72 nodes for data and task management
6928 nodes for MD simulations

How much is too much?

- GPUs are becoming so powerful that large numbers of atoms are needed to saturate performance
- For expensive potentials (e.g., SNAP), **around ~10K atoms per GPU**
- For cheap potentials (e.g., EAM), **~10M atoms per GPU**



GPUs are too powerful!

- **Good news** for expensive potentials
  - Parallel MD: **>300M** atoms for SNAP
  - ParSplice: **~10K** atoms per replica, **50% of peak performance if running 1K atom**

- **Bad news** for cheap potentials.
  - Parallel MD: **>300B** atoms for EAM
  - ParSplice: **~30M** atoms per replica, **1% of peak performance if running 1K atom**

- It is possible to oversubscribe GPUs, but only limited performance improvement with out-the-box methods

- Bad news for anything that doesn't require expensive potentials and/or millions of atoms!
  - 90% of what people currently do with MD

GPUs are too powerful!

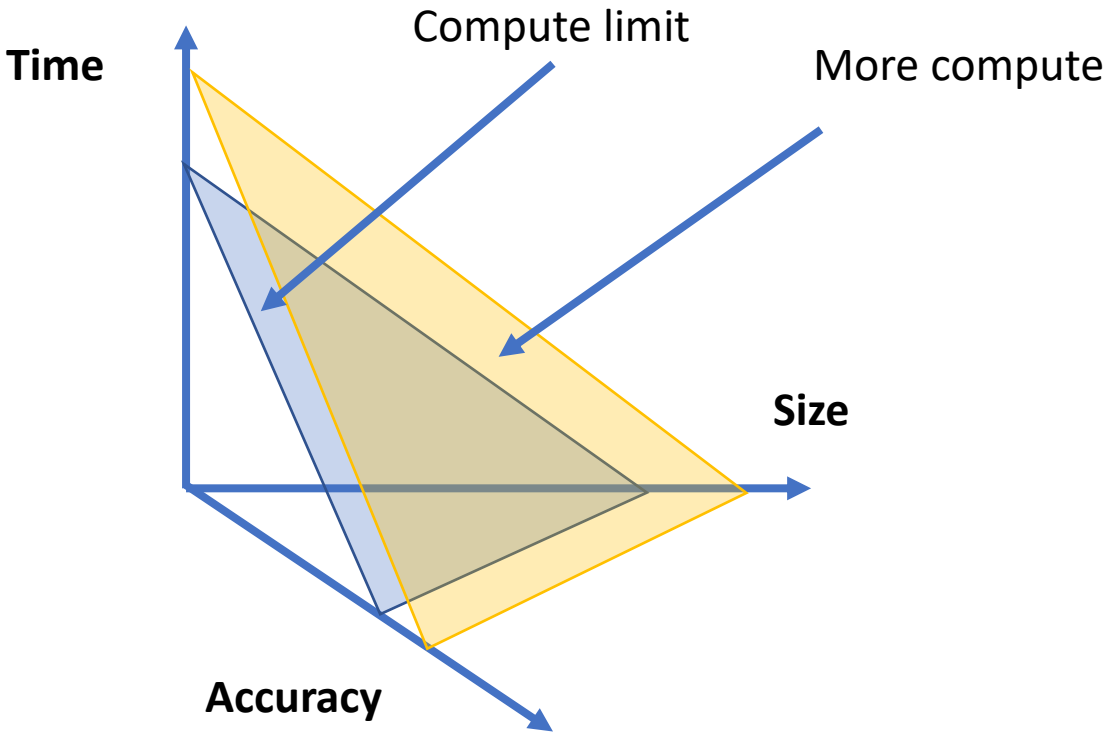
- Solution would be to enable MD codes to run many systems concurrently in a seamless fashion

- A single MD timestep would propagate **many systems** at once

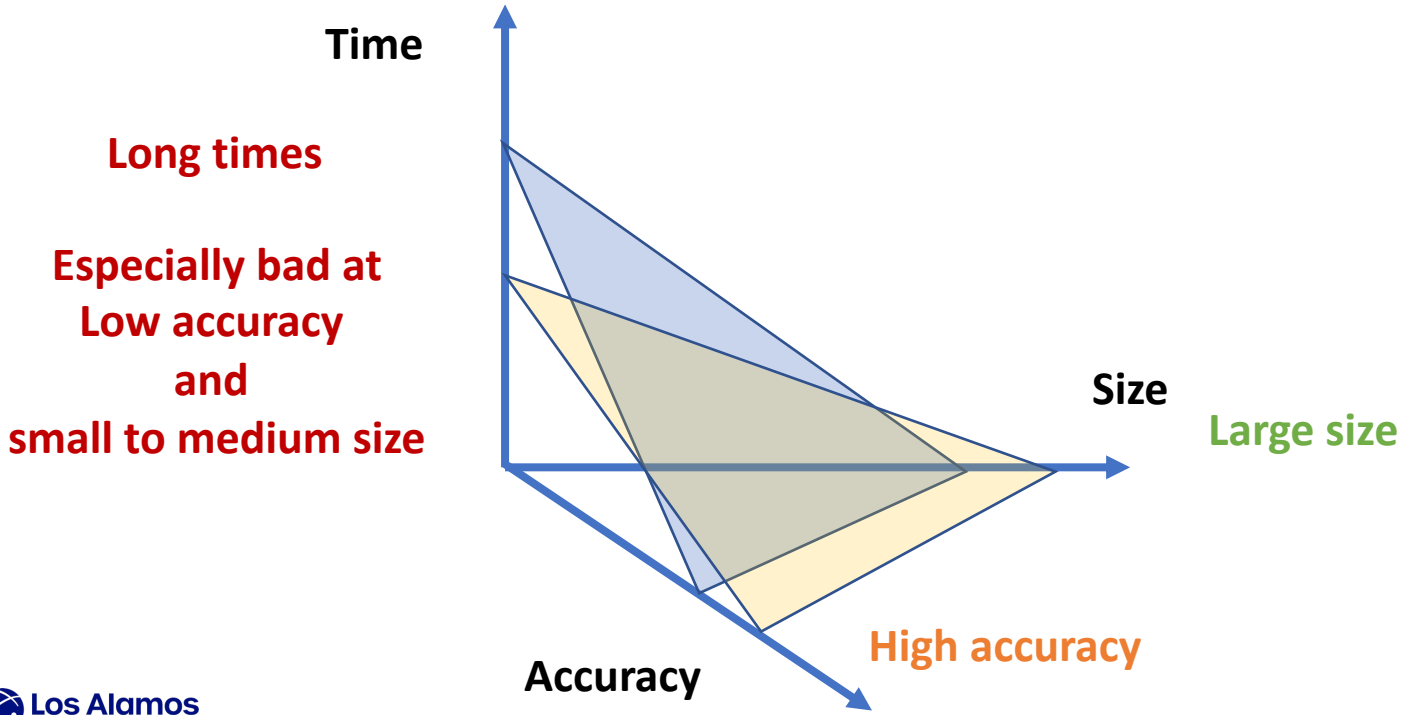
- Proof of concept (Lubbers and Mehta):
  - Concatenate all systems into a single list of atoms
  - Create a combined neighbors list, with potentially different simulation cell for each system
  - Compute forces all at once on the GPU: atoms from different systems don't see each other
  - Integrate all systems in lockstep

One would need to expose this capability in a way that is transparent to users and allow for the reuse of existing algorithms. Interesting **software engineering** problem!

What do we *want* out of exascale



What do we *get* out of exascale with standard methods



Conclusion/Outlook

- Large computers offers unprecedented opportunities for atomistic simulations, but also challenges
- The technology is pushing simulations towards more expensive models and larger simulations
- This is now true even at the single node level!
- Significant **methodological and software evolution** will be required to avoid the simulation space **moving away from scientifically relevant regimes**