Coupling the Time-Warp algorithm with a KMC framework for exact distributed simulations of heterogeneous catalysts

Dr Ilektra Christidi

Senior Research Software Developer

UCL Advanced Research Computing Centre

Computing Insight UK, 1-2 December 2022

Outline

- The *Zacros* Kinetic Monte Carlo package
- Time-Warp algorithm
- Performance
    - Parameter tuning
    - Scaling
    - Large system

The *Zacros* team

A long-standing team effort between UCL Chemical Engineering and RITS/ARC


Prof. Michail Stamatakis


Raz Benson


Giannis Savva


Ilektra Christidi


David Stansby


Roland Guichard


Srikanth Ravipati


Miguel Pineda


James Hetherington

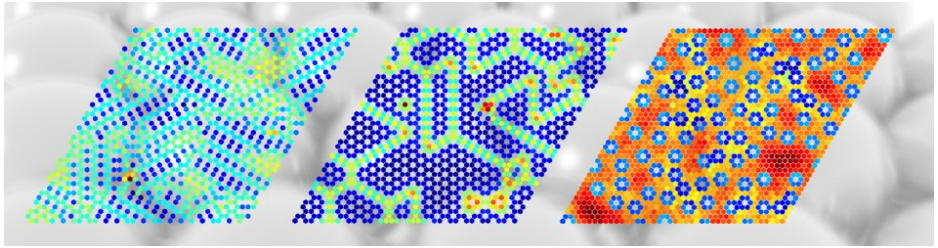

Jens NIelsen


Mayeul D'Avezac

Zacros

Zacros is a Kinetic Monte Carlo package for simulating molecular phenomena on catalytic surfaces.

- **A lattice with sites**
- **A queue of processes (adsorption, desorption, diffusion, reactions)**
  - **Ordered in time based on probability, calculated stochastically based on the energetics of each site**
- **Every KMC step, the most imminent process is executed, lattice occupancies updated, energetics recalculated, and the queue updated**



Limitations

- Large lattices, long KMC times → collective phenomena, eg. pattern formation.
- Need to parallelise, but KMC is serial in nature (which process happens now depends on processes in the past)
- Processes are local, therefore domain decomposition should be possible → need a way to handle processes and interactions across domains
    - The ***Time-Warp*** algorithm

The parallelisation scheme

An *optimistic Parallel Discrete Event Simulation*: keep progressing the KMC time until told otherwise.

- **Minimal synchronization required**

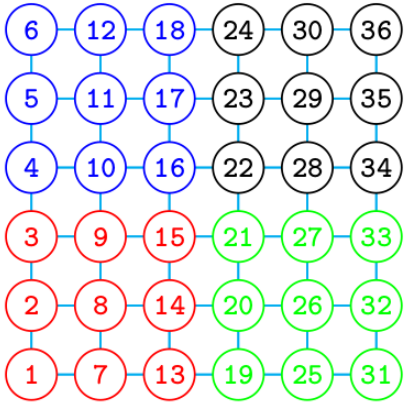
- **Decompose the lattice into subdomains, assign each to an MPI process**
- **Every subdomain progresses independently and holds its own local data structures**
- **Conflicts at the boundaries are communicated via point-to-point *messages* between neighbors asynchronously**
- **Causality violations resolved via a *rollback* procedure**
- **Global communications at fixed wall-clock time intervals to keep track of the *Global Virtual Time* (GVT)**

The Time-Warp algorithm

Domain decomposition

Events that need to be communicated between MPI processes:

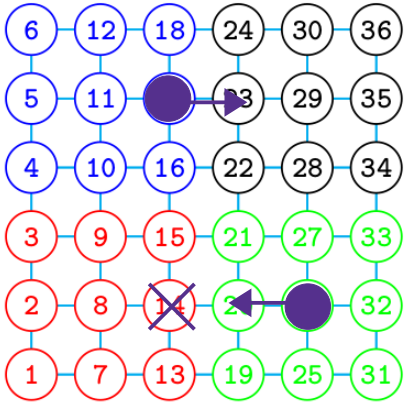
- **Particles across borders**
- **Interactions across borders**



Domain decomposition

Events that need to be communicated between MPI processes:

- **Particles across borders**
- **Interactions across borders**

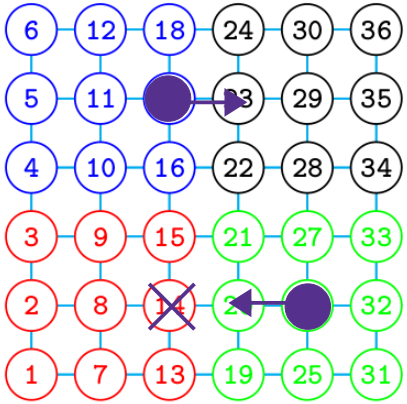


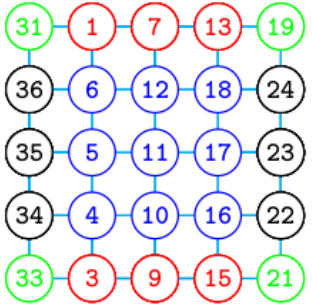
Domain decomposition

Events that need to be communicated between MPI processes:

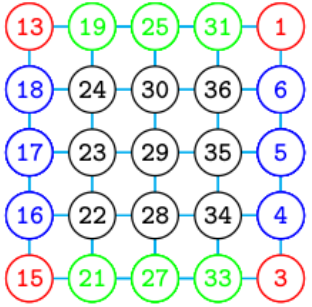
- **Particles across borders**
- **Interactions across borders**

=> Halos have to be large enough to capture those events

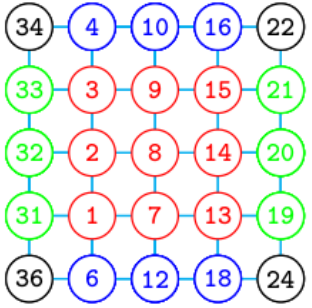




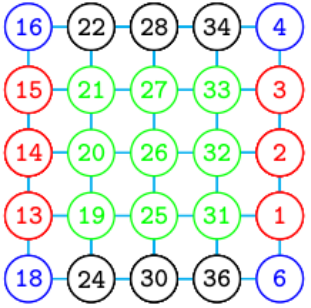
(a) PE-1



(b) PE-3



(c) PE-0



(d) PE-2

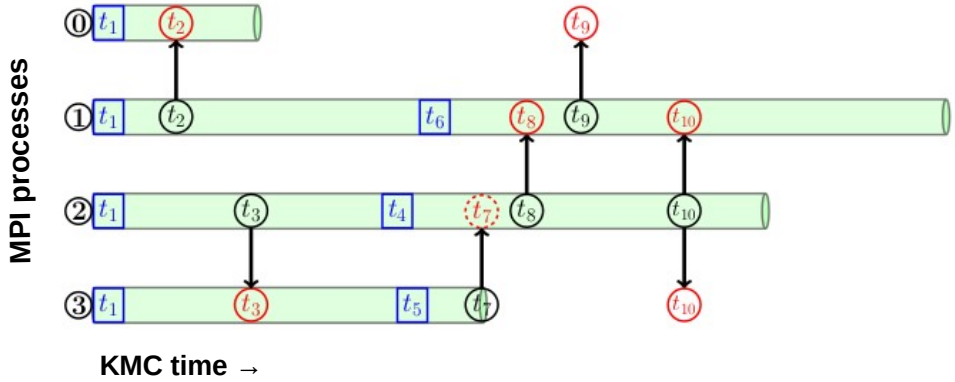
The Time-Warp algorithm

- **A *message* between neighboring MPI processes signals the receiver to add the messaged event to its own process queue at the right time.**
- **At every KMC step, the most imminent process can be a local one or a messaged one.**
- **If the received message is in the past of the local KMC time, it's a *causality violation*, and a *rollback* has to happen**

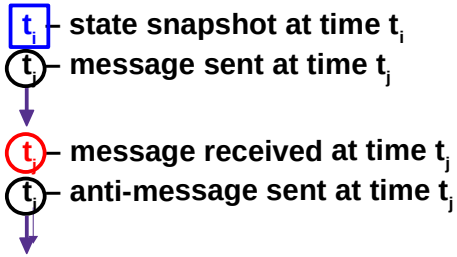
Data structures:
- **Message queue**
- **State queue**

The Time-Warp algorithm

- A *message* between neighboring MPI processes signals the receiver to add the messaged event to its own process queue at the right time.
- At every KMC step, the most imminent process can be a local one or a messaged one.
- If the received message is in the past of the local KMC time, it's a *causality violation*, and a *rollback* has to happen

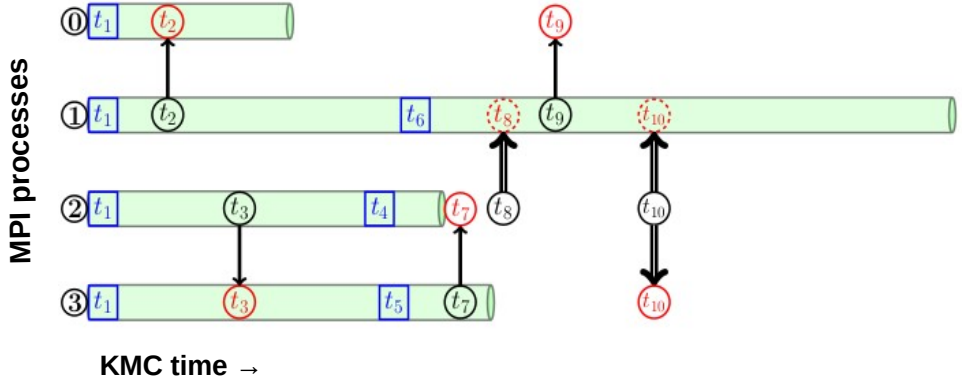


Data structures:
- Message queue
- State queue



Rollback

- A *message* between neighboring MPI processes signals the receiver to add the messaged event to its own process queue at the right time.
- At every KMC step, the most imminent process can be a local one or a messaged one.
- If the received message is in the past of the local KMC time, it's a *causality violation*, and a *rollback* has to happen
  - Re-instate a system state previous to the message timestamp



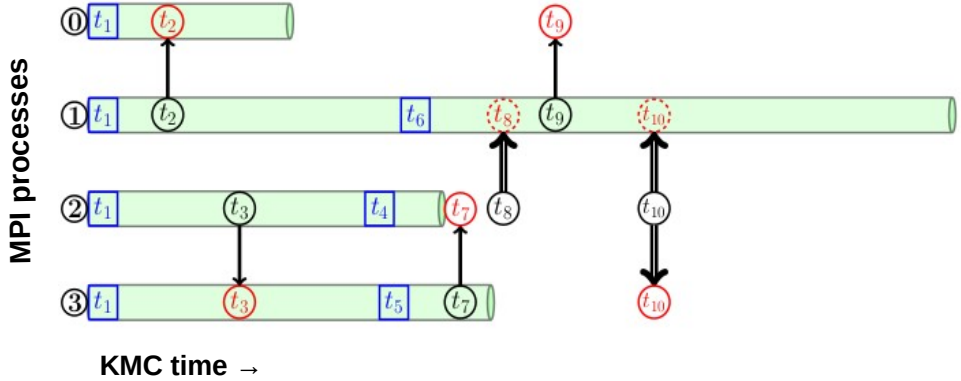
Data structures:
- Message queue
- State queue

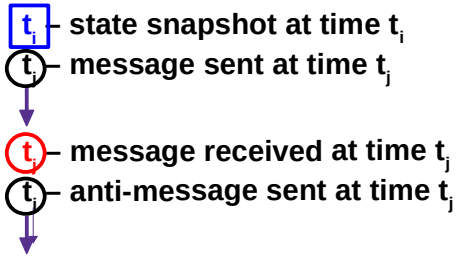
t_i – state snapshot at time t_i
t_j – message sent at time t_j
t_j – message received at time t_j
t_j – anti-message sent at time t_j

Rollback

- A *message* between neighboring MPI processes signals the receiver to add the messaged event to its own process queue at the right time.
- At every KMC step, the most imminent process can be a local one or a messaged one.
- If the received message is in the past of the local KMC time, it's a *causality violation*, and a *rollback* has to happen
  - Re-instate a system state previous to the message timestamp
  - Undo any messages sent to other MPI processes after that time, by sending them *anti-messages*

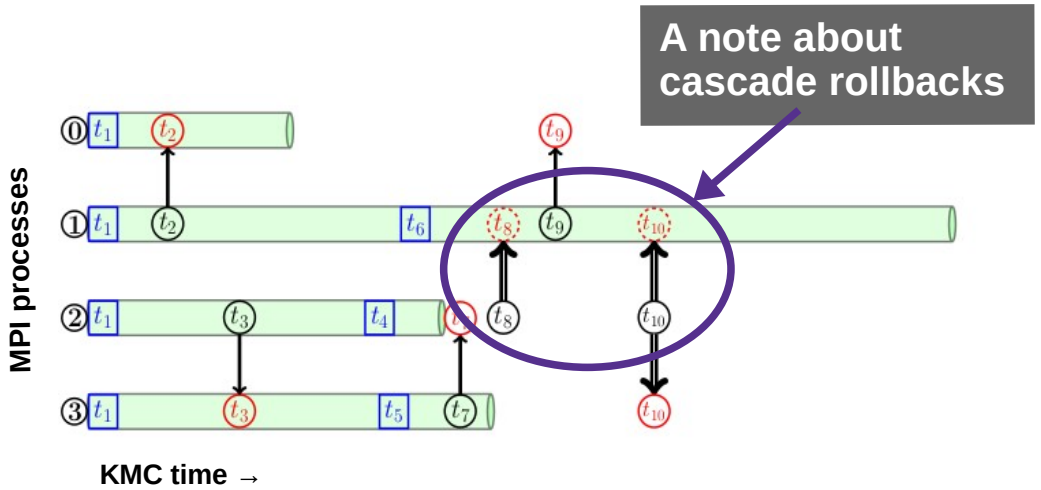
Data structures:
- Message queue
- State queue





Rollback

- A *message* between neighboring MPI processes signals the receiver to add the messaged event to its own process queue at the right time.
- At every KMC step, the most imminent process can be a local one or a messaged one.
- If the received message is in the past of the local KMC time, it's a *causality violation*, and a *rollback* has to happen
  - Re-instate a system state previous to the message timestamp
  - Undo any messages sent to other MPI processes after that time, by sending them *anti-messages*



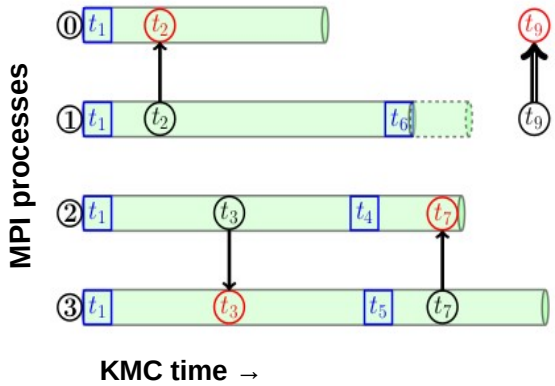
Data structures:
- Message queue
- State queue

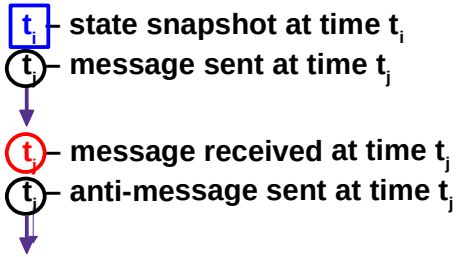
t_i – state snapshot at time t_i
t_j – message sent at time t_j
t_j – message received at time t_j
t_j – anti-message sent at time t_j

Rollback

- A *message* between neighboring MPI processes signals the receiver to add the messaged event to its own process queue at the right time.
- At every KMC step, the most imminent process can be a local one or a messaged one.
- If the received message is in the past of the local KMC time, it's a *causality violation*, and a *rollback* has to happen:
  - Re-instate a system state previous to the message timestamp
  - Undo any messages sent to other MPI processes after that time, by sending them *anti-messages*
  - Re-simulate the history, taking into account the received event

Data structures:
- Message queue
- State queue





Global communication

Need to know the *Global Virtual Time* (GVT) – minimum of KMC times of all MPI processes and the timestamps of any messages in transit

- **Clean up state and message queues from old entries that will not be used anymore (t<GVT)**
- **Decide when the run is over – GVT≥KMCtime(max)**

Collective communication happens at a regular wall-clock time interval.

Algorithm parameters

Time Warp is a memory hungry algorithm – KMC state snapshot queue can get large. Therefore, performance depends on

- **Memory available**
- ***Snapshot-taking interval*** **(δ_{snap}): how often a state snapshot is taken, in # of events**
    - **The more snapshots in the queue, the more efficient the rollbacks**
- ***GVT interval*** **($\Delta\tau_{GVT}$): how often the GVT is calculated, in s**
    - **Memory is cleaned up for re-use after every GVT calculation**

Optimum will be when the benefit of having enough snapshots in the state queue to perform efficient rollbacks, offsets the cost of taking more snapshots and/or global communications

→ Experimented with different state queues for efficient snapshooting:

- ***linked list*** **vs** ***vector***

Systems used for benchmarking

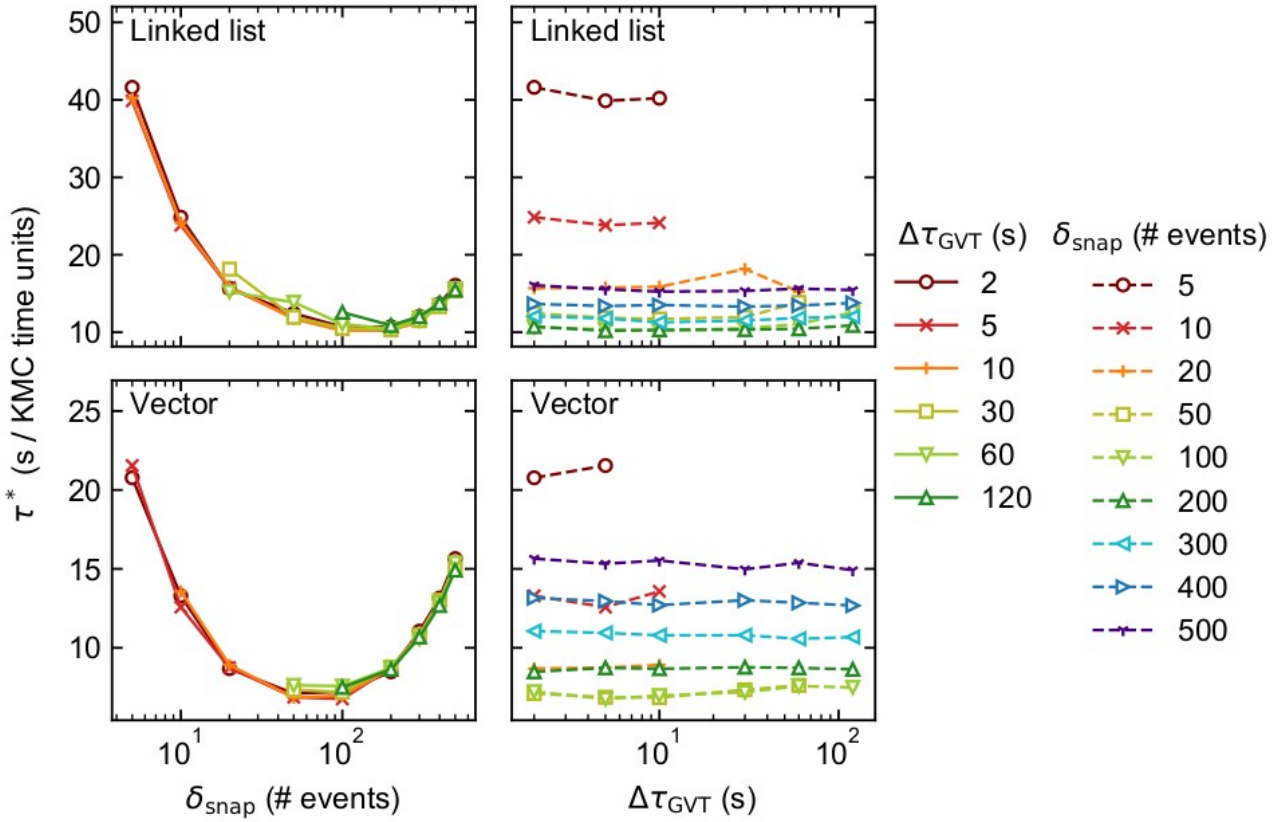
- System 1: adsorption, desorption, diffusion
  - Strong coupling between domains, due to particles crossing boundaries
- System 2: adsorption, desorption, nearest-neighbor lateral interactions
  - Weak coupling between domains, due to energetic clusters crossing boundaries
- System 3: several reactions (complex CO oxidation mechanism) and full energetic cluster expansion
  - Realistic system, with the strongest coupling between domains and large halos

Parameter studies

System 1

200x200 sites

4 MPI processes

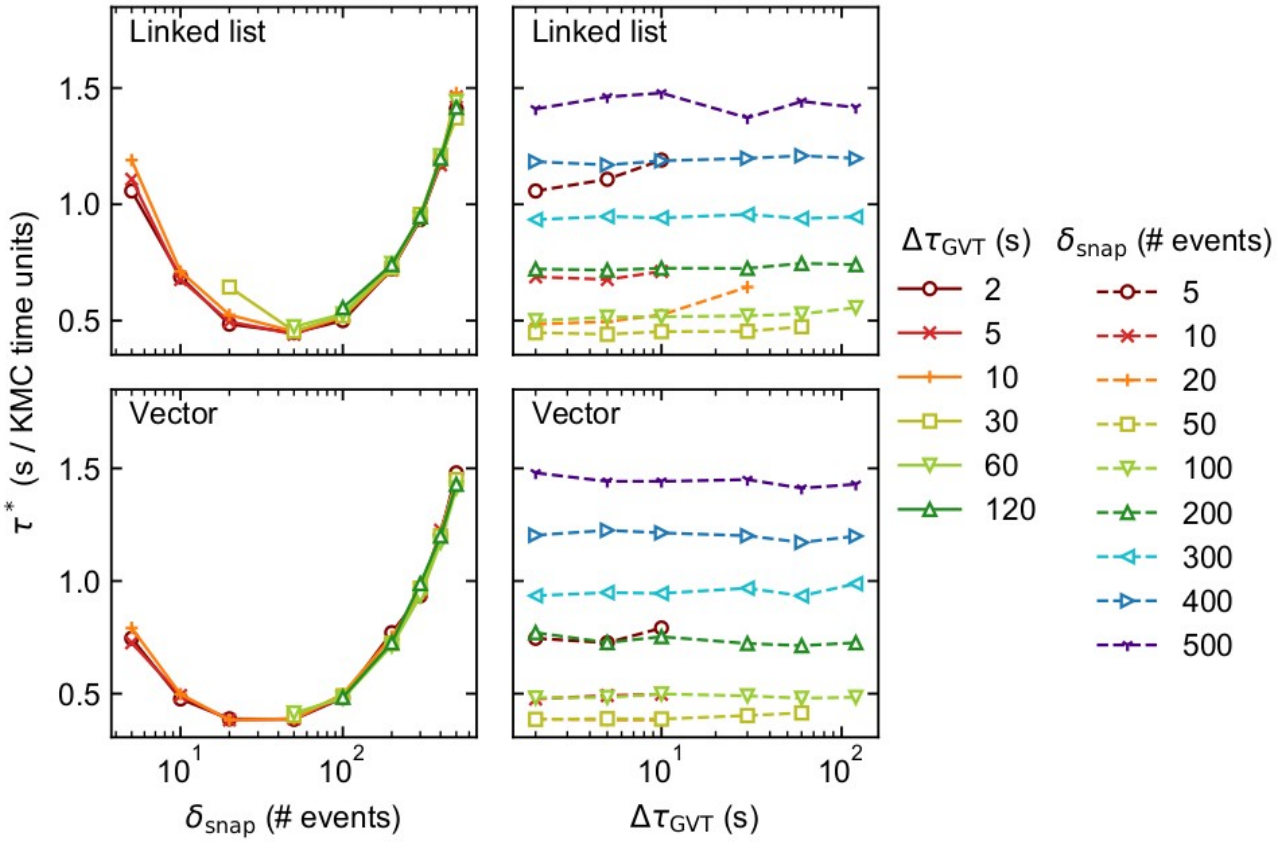


Parameter studies

System 2

200x200 sites

4 MPI processes



Parameter studies - conclusions

- **There is an optimum choice of δ_{snap} and $\Delta\tau_{GVT}$, though it varies by chemical system and number of MPI processes**
- **δ_{snap} affects performance much more than $\Delta\tau_{GVT}$**
- **Type of queue makes a difference**

→ Guidance to users to perform preliminary studies before production runs, to choose optimal parameters

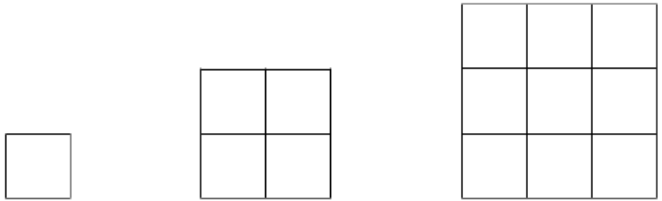
G. Savva, et al (2022). *Large-scale benchmarks of the Time-Warp/Graph-Theoretical Kinetic Monte Carlo approach for distributed on-lattice simulations of catalytic kinetics*. Under review.

Scaling studies

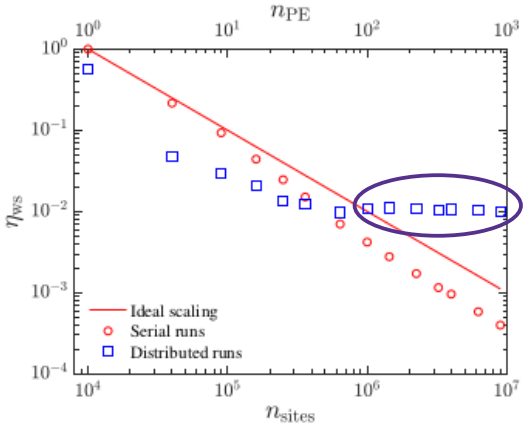
Weak scaling

$$\eta = \frac{t^*\left(n_{\text{sites}}\right)}{t^*\left(n_{\text{sites}}^{min}\right)}$$

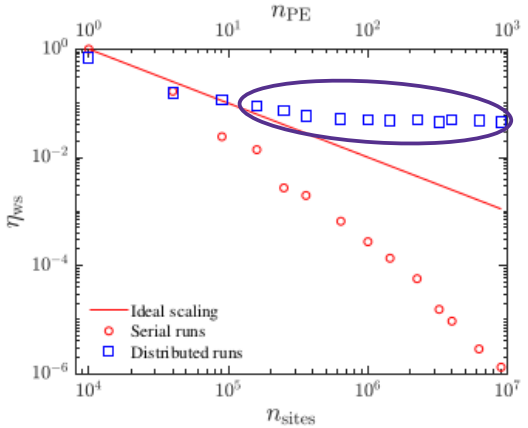
$$\eta_{\text{ws}} = \frac{t^*\left(n_{\text{sites}} : n_{\text{PE}}\right)}{t^*\left(n_{\text{sites}}^{min}\right)}$$



(a) 1×1 (b) 2×2 (c) 3×3



(d) System 1

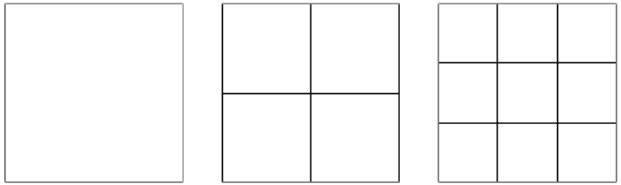


(e) System 2

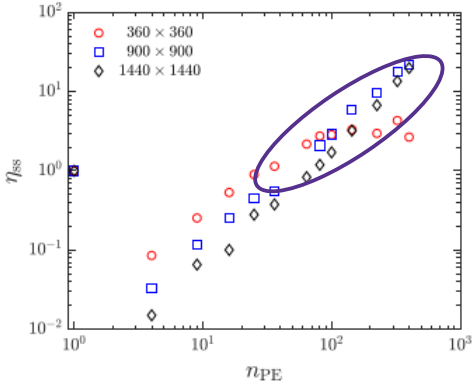
Scaling studies

Strong scaling

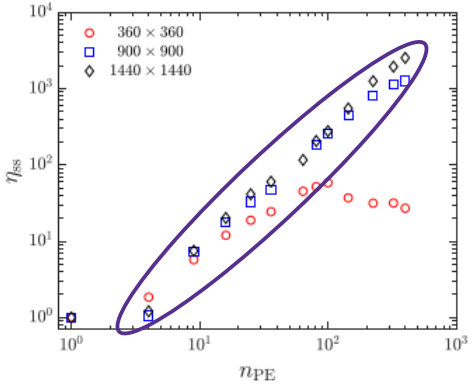
$$\eta_{\mathrm{ss}} = \frac{t^*\left(n_{\mathrm{sites}} : n_{\mathrm{PE}}\right)}{t^*\left(n_{\mathrm{sites}}\right)}$$



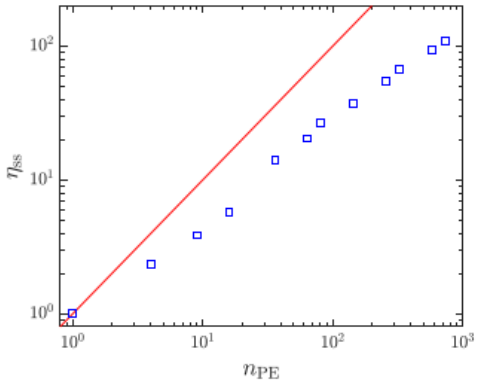
(a) 1×1 (b) 2×2 (c) 3×3



(d) System 1



(e) System 2



(a) System 3 - performance benchmarks

Scaling studies - conclusions

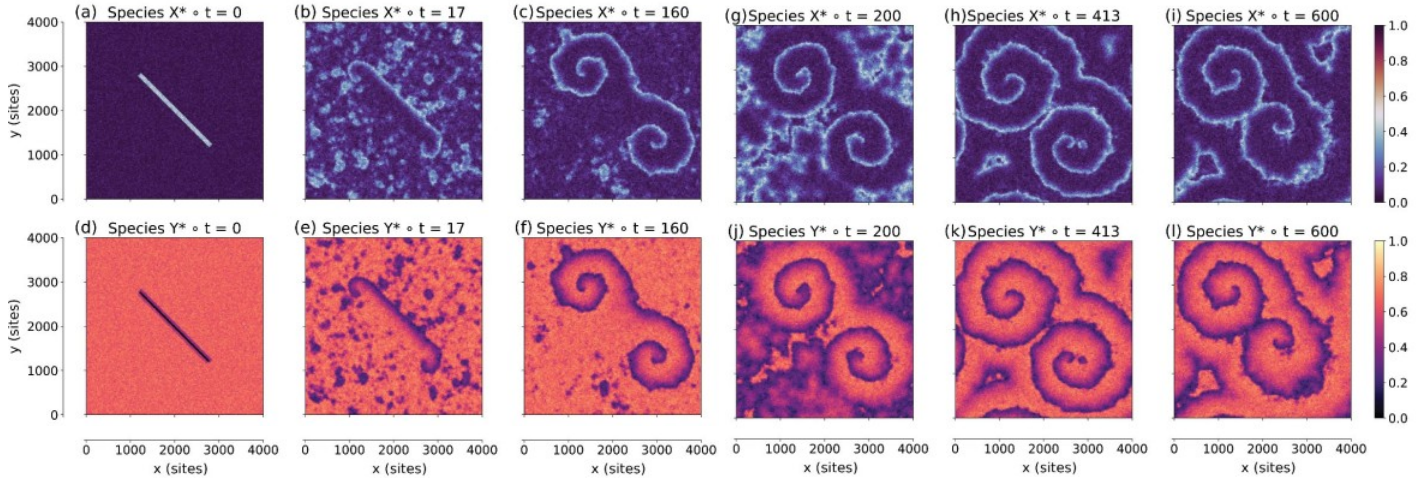
- **The Time Warp algorithm as implemented in Zacros can have significant overheads and memory requirements**
- **For large enough lattices (i.e. minimizing the relative size of halos) it scales well and out-performs serial KMC**

S. Ravipati, et al. (2022). *Coupling the Time-Warp algorithm with the Graph-Theoretical Kinetic Monte Carlo framework for distributed simulations of heterogeneous catalysts*. Comput. Phys. Commun. 270, 108148. (doi: 10.1016/j.cpc.2021.108148)

Performance on large system

Simulated a lattice-based variant of the Brusselator system of 16M sites on Thomas@UCL

- **25x25 = 625 MPI processes → x16 speedup**
- **40x40 = 1600 MPI processes → x36 speedup**
- **Total runtime: 620 KMC s == 38 days (~1.5yr of serial run)**
- **More than 1.6 trillion events*** **(*without the rollbacks)**



G. Savva, et al. (2022). *Exact Distributed Kinetic Monte Carlo Simulations for On-Lattice Chemical Kinetics: Lessons Learnt from Medium- and Large-Scale Benchmarks*. Under review.

Conclusions

- The implementation of Time-Warp into *Zacros* has allowed larger lattices to be simulated for longer KMC times, which was not possible with the serial algorithm
- Optimal parameters found and methodology developed
- Good scaling, depending on simulated system and number of MPI processes
- Large simulation performed successfully
- Further work on memory usage optimisation ongoing

Zacros

Advanced lattice-KMC made easy

For more info https://zacros.org/

Supported by ARCHER-eCSE10-08, ARCHER2-eCSE01-13, Leverhulme Trust RPG-2017-361, EU Horizon 2020 GA: 814416

Facilities used: Thomas@UCL, ARCHER2

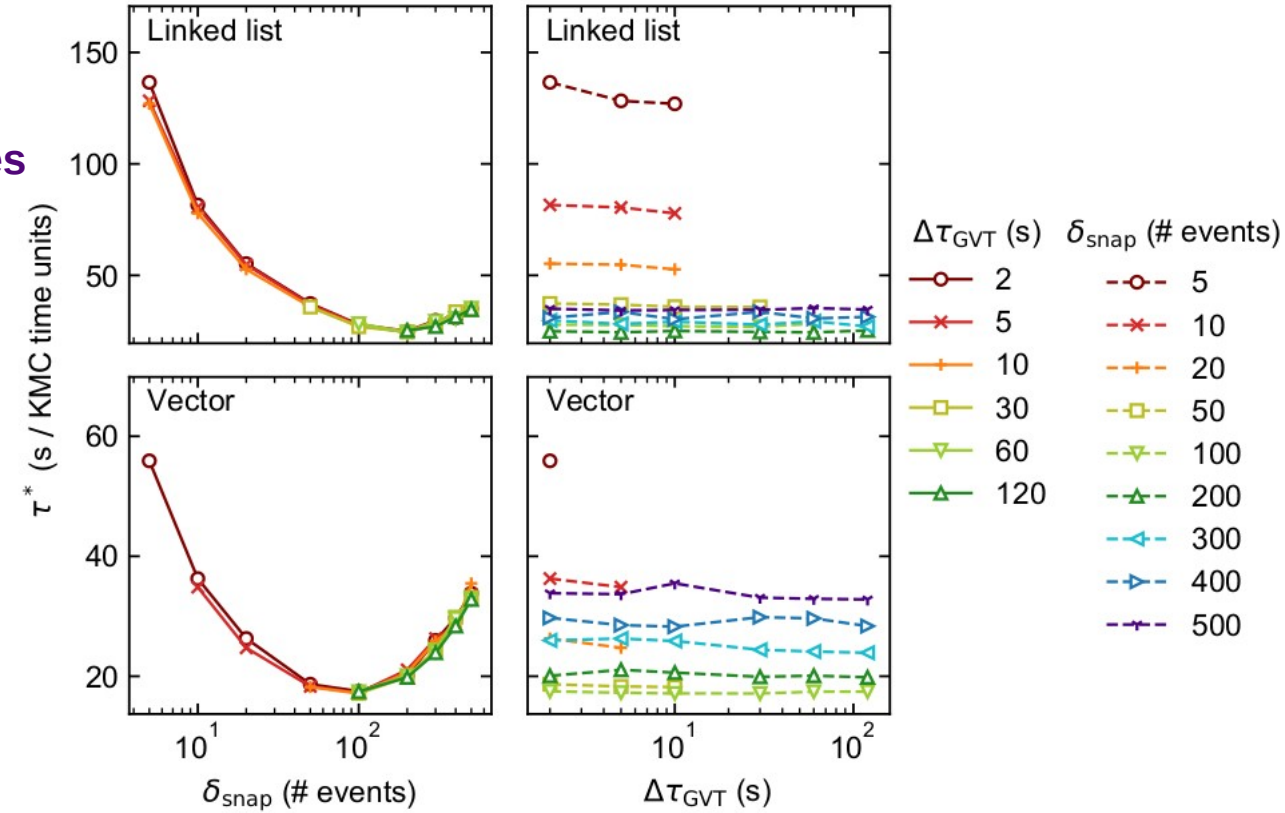
Extras

Parameter studies

System 1

1200x1200 sites

144 MPI processes

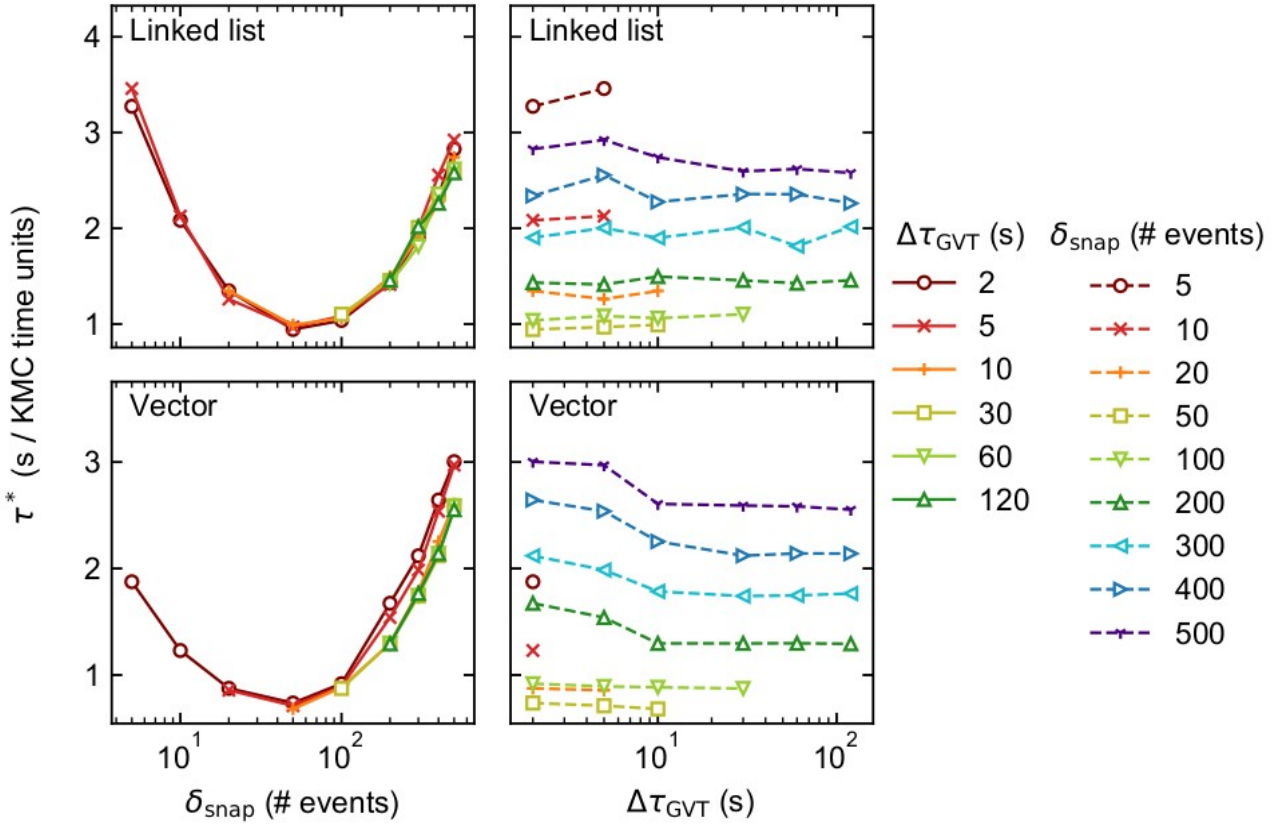


Parameter studies

System 2

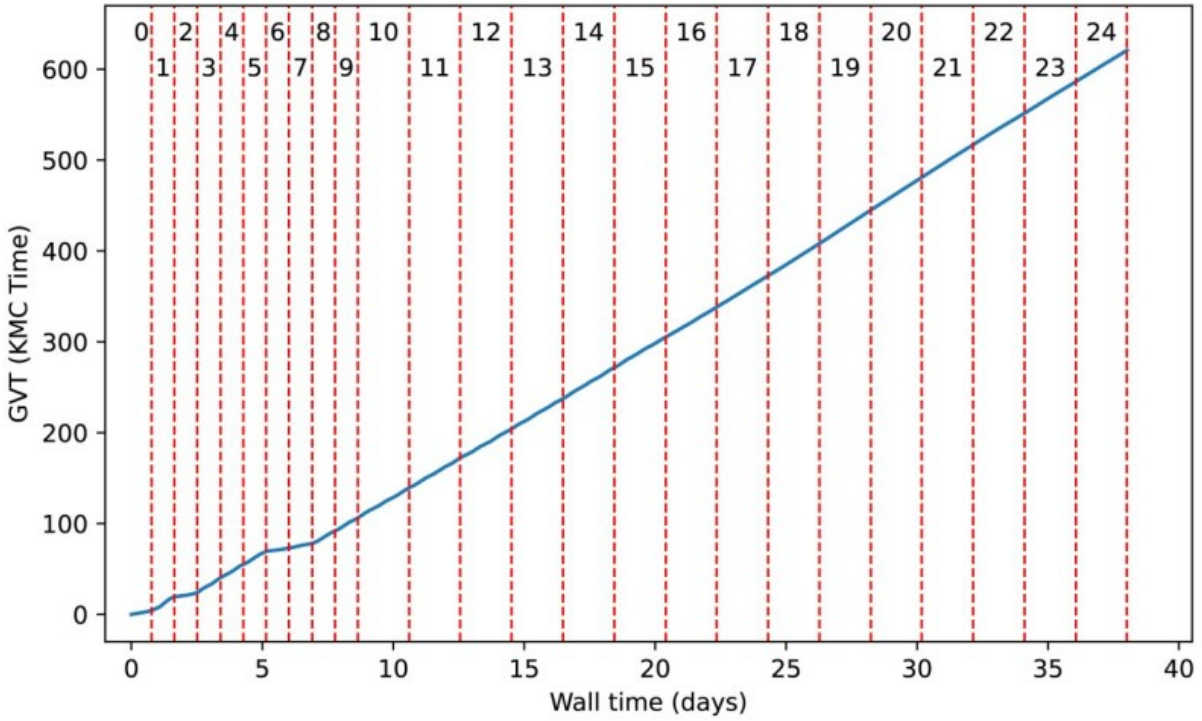
1200x1200 sites

144 MPI processes



Performance on large system

GVT progression in the 625 MPI processes run

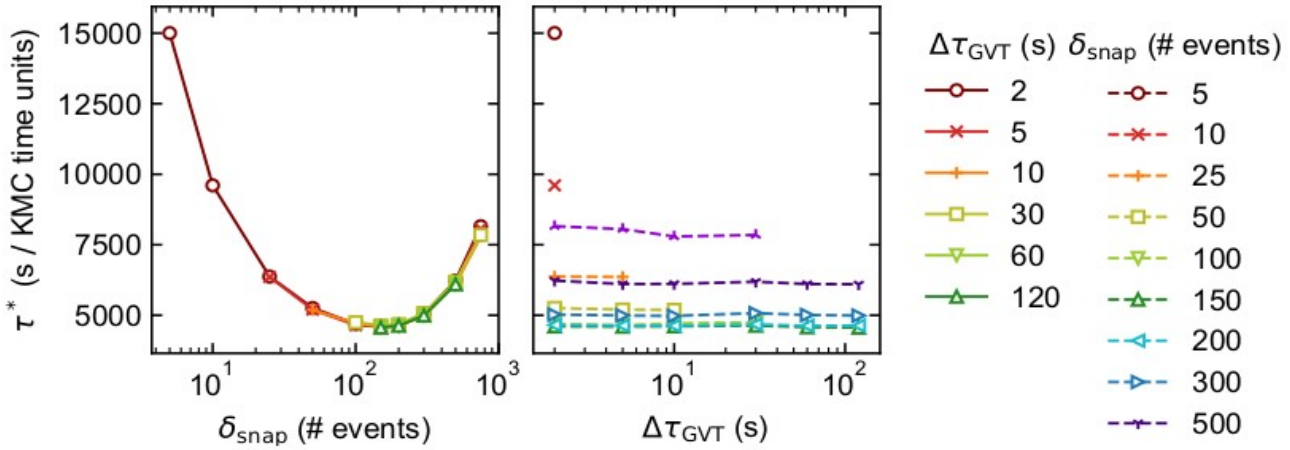


Parameter studies

Brusselator

1200x1200 sites

625 MPI processes



State queue size management

A KMC state snapshot is taken every certain number of events, provided by the user (*snapshot-taking interval*).

A fixed, user-defined amount of memory is available for the state queue throughout the run.

When memory gets filled up, the state queue is *sparsified* (every other stored state is deleted) to free up space, and the snapshot-taking interval is increased (to delay it being filled up again).

Rollbacks are less efficient when the state queue is sparsified, but at least we don’t run out of memory.

=> Even with sub-optimal parameters chosen, the simulation will not crash, it will just not be as performant

State queue data types

- Doubly-linked list
  - *Pros*: flexible and straightforward to incorporate in algorithm
  - *Cons*: inefficient due to frequent allocations/deallocations of large objects (state snapshots) – every time we take snapshot, rollback, or cleanup
- Vector
  - *Pros*: all allocations happen in the beginning → efficient
  - *Cons*: cannot accommodate variable sized snapshots (so not appropriate for dynamic arrays)
- Optimised doubly-linked list
  - *Pros*: all of doubly-linked list, plus same efficiency as vector – instead of deallocating snapshots at rollback or cleanup, move them to end of queue and re-use them
  - *Cons*: cannot accommodate variable sized snapshots
- Variable-element doubly-linked list
  - *Pros*: same as optimised doubly-linked list, but can also accommodate variable sized snapshots
  - *Cons*: not as efficient as vector/optimised doubly-linked list (some allocations/ deallocations still happen until final size of dynamic arrays is reached)

Further optimisations

- **Investigate memory usage optimisation: implement dynamic arrays**
  - **Preliminary results: memory utilisation for the process queue went up by 25% (from 65% to 80%)**
- **Optimise KMC state snapshot queue: want the flexibility of linked list, but the performance of vector queue**
  - **“optimised linked list” and a variant of it for dynamic arrays inside each state**
  - **Preliminary results: optimised linked list performs as good as vector, variable element linked list performs ok**