Coupling the Time-Warp algorithm with the Graph-Theoretical Kinetic Monte Carlo framework for distributed simulations of heterogeneous catalysts

Srikanth Ravipati[a], Giannis D. Savva[a], Ilektra-Athanasia Christidi[b], Roland Guichard[b], Jens Nielsen[b], Romain Réocreux[a], Michail Stamatakis[a,*]

[a] *Thomas Young Centre and Department of Chemical Engineering, University College London, Torrington Place, London WC1E 7JE, United Kingdom*
[b] *Research Software Development Group, Research IT Services, University College London, Torrington Place, London WC1E 6BT, United Kingdom*

Abstract

Despite the successful and ever widening adoption of kinetic Monte Carlo (KMC) simulations in the area of surface science and heterogeneous catalysis, the accessible length scales are still limited by the inherently sequential nature of the KMC framework. Simulating long-range surface phenomena, such as catalytic reconstruction and pattern formation, requires consideration of large surfaces/lattices, at the μm scale and beyond. However, handling such lattices with the sequential KMC framework is extremely challenging due to the heavy memory footprint and computational demand. The Time-Warp algorithm proposed by Jefferson [*ACM. Trans. Program. Lang. Syst.*, 1985. 7: 404-425] offers a way to enable distributed parallelization of discrete event simulations. Thus, to enable high-fidelity simulations of challenging systems in heterogeneous catalysis, we have coupled the Time-Warp algorithm with the Graph-Theoretical KMC framework [*J. Chem. Phys.*, 134(21): 214115; *J. Chem. Phys.*, 139(22): 224706] and implemented the approach in the general-purpose KMC code *Zacros*. We have further developed a "parallel-emulation" serial algorithm, which produces identical results to those obtained from the distributed runs (with the Time-Warp algorithm) thereby validating the correctness of our implementation. These advancements make *Zacros* the first-of-its-kind general-purpose KMC code with distributed com-

*Corresponding author.
E-mail address: m.stamatakis@ucl.ac.uk

puting capabilities, thereby opening up opportunities for detailed meso-scale studies of heterogeneous catalysts and closer-than-ever comparisons of theory with experiments.

Keywords: Kinetic Monte Carlo, Lattice, Time-Warp algorithm, Catalysis, Materials Science, Distributed simulation

---

1. Introduction

Kinetic Monte Carlo (KMC) simulations have been successfully used in a variety of research fields [1, 2, 3, 4, 5, 6, 7, 8, 9] to predict the dynamic properties of materials and gain fundamental insights into the underlying microscopic phenomena. Heterogeneous catalysis is one such field that has seen a wide adoption of on-lattice KMC simulations, in which the catalyst surface is represented as a lattice [10, 5, 11, 12, 13, 14, 15, 16] (from here onwards, KMC refers to on-lattice KMC). Adsorption, desorption, diffusion, and reactions are considered as the elementary events of interest, whose rates can be obtained from first-principles. To this end, density functional theory (DFT) calculations, which strike a practical balance between accuracy and computational expense, are used in tandem with transition state theory approaches. Properties of interest, such as activity, selectivity and stability, which serve as metrics in the identification of promising catalysts, can be calculated by sampling the realizations (stochastic trajectories) obtained by KMC simulations. In addition, such simulations can yield a fundamental understanding of the link between the underlying microscopic mechanism and the resultant macroscopic observables. Such a multi-scale understanding aids in the development of predictive models, which are exceptionally promising in designing better catalysts, in line with the "rational catalyst design" vision. In turn, the accurate prediction of catalytic performance metrics (activity, selectivity, stability) is critical to the design of reactors and the improvement of the overall performance of a chemical process. Indeed, the increasing interest in KMC simulations for catalysis and chemical reaction engineering applications has motivated the development of general-purpose software, such as *Zacros* [17, 18] (`https://zacros.org/`), a versatile, high-fidelity Graph-Theoretical (GT)-KMC code. SPPARKS [3], CARLOS [19], kmos [20], and KMCLib [21] are also examples of popular KMC codes available.

The complexity of a KMC simulation varies depending on the specific features of the diverse elements involved in the model: lattice structure,

binding configurations of adsorbates, reaction mechanism, and adsorbate-adsorbate lateral interactions. Thus, systems with e.g. multidentate adsorbates, long-range lateral interactions and complicated reaction pathways tend to be computationally expensive. Besides, the KMC framework is inherently sequential, in the sense that lattice events are executed one after another based on their priority (an event with lower KMC time has a higher priority). For these reasons, KMC simulations of heterogeneous catalysts are limited to small lattices, typically on the order of 10 nm $\times$ 10 nm [22, 23].

There have been numerous efforts in improving the performance of KMC simulations, and this is still a topical research area. Approximate methods, such as scaling down rate-constants [24, 25, 26, 27, 28], lumping states together and applying theory of absorbing Markov chains [29, 30, 31], and implementing fast species redistribution [32] have been used to alleviate the time-scale separation challenge, whereby certain "uninteresting" frequent events, such as diffusional hops, dominate the simulation effort, at the expense of "significant" events such as reactions. On the other hand, exact efficient algorithms/implementations [33, 34, 17, 35, 18, 36, 37, 38] have improved the performance of KMC simulations without the loss of accuracy.

Although KMC simulations of small to moderately sized lattices can yield acceptable-quality estimates of the aforementioned observables, understanding phenomena involving long-range spatial variations, such as catalytic surface reconstruction and pattern formation, necessitate simulations of large systems. In such scenarios, the required size of the simulated lattice would be dictated by the characteristic wavelength of the pattern, which could be in the order of μm to mm. For instance, in the context of reconstruction, oscillations and pattern formation have indeed been observed at the μm scale [39], and lead to instabilities that must be controlled at the reactor scale [40]. Thus, in addition to the previously noted complexities involved in a KMC simulation, the need to consider large lattices poses extra challenges in terms of memory footprint and computational time. At present, it is infeasible to simulate realistic chemistries on catalytic surfaces that span μm in size within the sequential KMC framework, and distributed parallelization is a way to address this challenge.

Different approaches, such as the synchronous [41], the synchronous relaxation [42], the optimistic Time-Warp [43], the optimistic synchronous relaxation [44, 45], and the semi-rigorous synchronous algorithms [46, 47] are available for the distributed parallelization of discrete event simulations. At a high level, these approaches entail appropriate domain decomposition and

event execution protocols that either avoid boundary conflicts via synchronization, or employ rollbacks and re-simulations to correct for any causality violations arising from such conflicts. In the latter approach, provisional KMC trajectories are progressively amended and, once validated to be mutually consistent, are finally incorporated into the global simulation history. This approach is reminiscent of the more general concept of "software transactional memory" [48], by which a processor performs modifications to shared memory, at first independently of other processors. Then, these transactions are either validated and committed to memory, or aborted, in which case any changes are rolled-back and retried.

In terms of general software implementations in the computational catalysis and materials science fields, an approximate approach similar to the semi-rigorous synchronous sub-lattice algorithm [46, 47] has been implemented in SPPARKS, and used mainly for materials science applications [49, 50, 51, 52]. Additionally, the 'Catalysis' module of SPPARKS [53], can be used to perform distributed KMC simulations of heterogeneous catalysts. Moreover, SPOCK [54] is an exact KMC implementation based on the ROSS [55] discrete event simulator which incorporates Time-Warp features. SPOCK has been shown to scale well for model problems in 3D, capturing lattice gas dynamics and grain growth [54]. However, building custom models in SPOCK is somewhat complicated and requires the user to supply code for forward and reverse execution of events (the latter to be used in rollbacks). In addition, neither code appears to provide a rigorous mechanism for validating the correctness (or accuracy) of simulated runs, thereby safeguarding against coding errors or undesirable approximation errors. It thus emerges that, even though scalable simulation algorithms at the electronic and atomistic levels are already at a mature stage (see e.g. Ref. [56]), distributed mesoscopic scale simulation approaches are still at their infancy. To our knowledge, a validated scalable implementation of an exact general-purpose KMC approach for simulations of heterogeneous catalysts, is still lacking.

To fill this gap, we have coupled the graph-theoretical KMC framework with the optimistic Time-Warp algorithm and have implemented the approach in *Zacros*. In this approach, the lattice is decomposed into domains that are assigned to different processing elements (PEs) (each of which may be handled by a single core or involve several threads). Each PE executes the KMC algorithm for the assigned domain and communicates with the neighbors, if necessary. Thus, elementary events that happen at the interior of the lattice (far from domain boundaries), are handled privately and asyn-

chronously, while events close to the boundaries have to be communicated as appropriate. Communication among processes is handled using the message passing interface (MPI) framework (`https://www.mpi-forum.org/`). Due to the asynchronous execution of "internal" events, each PE follows its own simulation time; thus, when "boundary events" happen, causality violations may arise. Such violations are resolved using the state-saving, rollback, and anti-message protocols of the Time-Warp algorithm. Additionally, the necessary protocols for the computation of the global virtual time (the collective KMC time of all PEs) and the termination of the distributed run are implemented.

Crucially, a fundamental constraint underpinning the Time-Warp algorithm, enables us to develop a simple variant of the serial KMC algorithm that enables the validation of our implementation. For our case of temporal point processes, involving events that happen instantaneously after some random inter-arrival time (quiescence time), the constraint can be stated as follows: if an event A causes an event B, then event A must be scheduled and executed *before* (in real-time terms) event B. On this basis, we develop a "parallel-emulation" scheme, i.e. a serial KMC simulation protocol that utilizes the deviates generated by the random number generator in the same order as in the distributed run. Thus, the results of the parallel-emulation runs must be identical to those of distributed runs (down to the stochastic fluctuations), enabling meaningful comparisons to be made.

The rest of the paper is organized as follows. The implementation of the different Time-Warp simulation components and protocols is discussed in detail in section 2: Methodology. The parallel-emulation method is presented in section 2.9: Parallel-Emulation. Subsequently, the validation of our implementation and the performance benchmarks (in terms of both weak- and strong-scaling) are presented and discussed in section 3: Results and Discussion, followed by section 4: Summary and Conclusions.

2. Methodology

In this section, starting with a brief description of the sequential KMC framework, we discuss the coupling of GT-KMC and Time-Warp. We examine in detail the necessary protocols for domain decomposition, communication of boundary-events, resolution of causality violations, computation of global virtual time, and termination of the distributed run.

2.1. Kinetic Monte Carlo framework and parallelization strategy

A KMC simulation of a surface chemistry requires three inputs: a lattice representation of the surface, a set of energetic patterns (also referred to as clusters/figures) describing lateral interactions among adsorbates, and a reaction mechanism consisting of elementary events (also referred to as reaction patterns). In the GT-KMC framework, the lattice structure, energetic patterns, and reaction patterns are represented as graphs, thereby making it possible to capture complex entities in a versatile way. Before the start of the KMC run, all input is parsed and some preparatory operations take place, e.g. the lattice state is initialized with all sites empty (or with a given initial state if desired), and a random number generator (used for the calculation of the random times of event occurrence) is provided with an initial seed.

Upon commencement of the KMC run, all possible instances of the elementary events on the lattice are detected and added to the process queue (*procQueue*). The latter is essentially a complete "catalogue" of all adsorption, desorption, diffusion and reaction events that are possible given a lattice state and a reaction mechanism, and is kept up to date throughout the KMC run. Finding instances of an elementary event on the lattice involves mapping sites of the corresponding reaction pattern to lattice sites, so that the connectivity and site occupancy on the lattice are as defined in the initial state of the reaction (this is done by solving subgraph isomorphism problems as explained in Stamatakis and Vlachos [17]). This procedure is repeated for all the reaction patterns prescribed in the reaction mechanism, and the detected lattice event instances are added to *procQueue*.

In addition, *procQueue* also stores the "absolute" KMC time for the occurrence of the each event, which is calculated as the sum of the current KMC time and the inter-arrival time of the KMC event (also known as quiescence or waiting time). For time-independent reaction rates (for instance, when the temperature of the system is constant), the pertinent inter-arrival times are generated as exponential deviates, with rate parameters equal to the events' rate constants. The latter may be influenced by the presence of spectator species in the neighborhood of the event, which exert lateral interactions to the reactants. Within the GT-KMC, such effects are captured by considering environment-dependent activation energies, parameterized by Bronsted-Evans-Polanyi equations [57, 58, 18]. Computing the effect of interactions requires the detection of energetic patterns, which uses approaches similar to the detection of reaction patterns (subgraph isomorphism). The reaction rate constant of the event is then obtained using the Eyring equa-

tion [59] following standard transition state theory approximations. Detailed discussions on the computation of rates and occurrence times of events are available in our previous publications [17, 5, 11, 37].

In a sequential KMC run, the algorithm first finds the most imminent event listed in *procQueue* (i.e. the one that results in the smallest advancement of KMC time). A sequence of operations (outlined in Algorithm 1) handle the update of the lattice state, the pertinent data-structures and the KMC time. Thus, the removal of energetic patterns and reaction events whereby the reactants participate, happens first, followed by the removal of the reactants themselves from the lattice. Subsequently, the new energetic interactions are added to the pertinent data-structure. The algorithm proceeds with the detection of the new events that can happen, which are added to *procQueue* (note that for the correct calculation of the kinetic constants the energetic patterns must have already been detected and stored). In the presence of lateral interactions, the rate constants of events in the neighborhood of the just-executed reaction event must be updated, since changes in the site occupancy will result in different interaction terms pre- versus post-reaction. Finally, the KMC time and the step counter are advanced. The execution of events results in the generation of a sequence of several lattice states. Observables of interest, such as species coverages or catalytic activity, are obtained as averages over these states.

Algorithm 1: Sequential kinetic Monte Carlo algorithm

Data: Lattice, energetic clusters/figures, reaction mechanism, temperature, gas phase pressure and composition.
Result: A trajectory of lattice states.

```
1  begin
2      Initialize the lattice state LattState with all sites empty;
3      Initialize globClusterEnerList with all lattice instances of
         energetic patterns;
4      Initialize procQueue with all possible lattice events, including
         occurrence times;
5      while t_KMC ≤ t_KMC^final ∧ curr_step ≤ n_steps do
6          Find the most imminent event E_next among those in
             procQueue;
7          Find the lattice adsorbates which participate in E_next as
             reactants and store them in LattReacE_next;
8          Find the energetic patterns in which the adsorbates of
             LattReacE_next participate and remove them from
             globClusterEnerList;
9          Find the elementary events in which the adsorbates of
             LattReacE_next participate and remove them from
             procQueue;
10         for A in LattReacE_next do
11             if A is referenced in any existing energetic or reaction
                 pattern then
12                 Raise a fatal error and stop;
13             else
14                 Remove A from the lattice;
15             end
16         end
17         Add the products of E_next to the lattice;
18         Detect the new energetic patterns in which the products of
             E_next can participate and add them to GlobClusterEnerList;
19         Detect the new lattice events in which the products of E_next
             participate and add them to procQueue;
20         Update the rates and the occurrence times of affected events
             that are in the neighborhood (region of influence) of E_next;
21         Set t_KMC to the occurrence time of E_next; increment curr_step
             by 1;
22     end
23 end
```

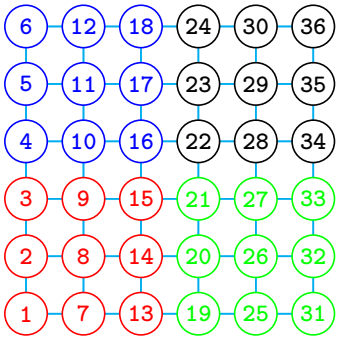


Figure 1: Decomposition of a 6×6 square lattice (36 sites in total) into four 3×3 domains (each with 9 sites). The sites of a domain are highlighted in the same color. Different colors represent different domains assigned to different PEs.

In a distributed run, the whole lattice is decomposed into subdomains, each of which is assigned to a PE (a processing element (PE) may be a CPU core, or, more generally, an MPI process). A random number stream is instantiated by each PE, either with the use of a distinct initial seed per PE for development/testing purposes, or, more rigorously, with appropriate methods that yield multiple disjoint sub-streams from a long random sequence, e.g. via the "jump-ahead" approach [60]. In addition to the operations just noted for sequential KMC simulations, a distributed KMC run includes additional protocols to handle the communication of boundary-events (sending and receiving messages), appropriate actions in response to a received message, the resolution of causality violations, the computation of global virtual time, and the collective termination of the run. The pertinent operations rely on two additional priority-sorted containers, a message queue (*messgQueue*) and a state queue (*stateQueue*). The former stores all incoming and outgoing messages, while the latter stores "KMC states" which are obtained at regular intervals and support the process of resolving causality violations through roll-backs. In the following, we discuss the pertinent datastructures and algorithmic components in detail.

2.2. Domain decomposition

In our implementation, we are dealing with periodic 2D lattices, which can be generated by tiling a unit cell N_α^C and N_β^C times along the two unit cell vectors α and β, respectively. We choose to work with subdomains encompassing an equal number of unit cells along the α and β directions,

since we would like to spread the load of communication due to boundary events as equally as possible along the neighbors. As will become clear later, the average frequency of message exchange between PEs depends on the number of sites within their common boundary; the constraint just noted, ensures the same number of sites for boundaries on different sides. Thus, in the example of fig. 1 each unit cell is a square containing one site (for simplicity unit cells are not drawn in the figure), and the lattice contains $6 \times 6 = 36$ sites in total. In this example, the lattice was partitioned into four 3×3 domains, but it could also be partitioned into nine 2×2 subdomains, in line with the aforementioned constraint (on the other hand, note that a partitioning into six 2×3 subdomains would violate this constraint).

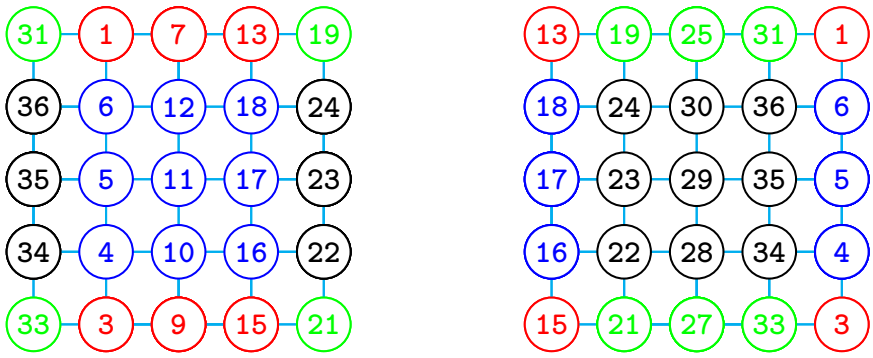
In general, if we need to partition the $N_\alpha^C \times N_\beta^C$ simulation cell into a given number of PEs (M_{tot}^P), we need to find the optimal "PE configuration" $M_\alpha^P \times M_\beta^P$, which makes use of as many PEs as possible. To this end, we can define the number of unit cells in the "side" of the subdomain, SZ, such that the subdomain will contain $SZ \times SZ$ unit cells in total, and then try to solve:

$$\begin{aligned} minimize \quad SZ \in \aleph \quad subject\ to: \quad N_\alpha^C &= M_\alpha^P \cdot SZ \\ N_\beta^C &= M_\beta^P \cdot SZ \\ M_\alpha^P \cdot M_\beta^P &\leq M_{tot}^P \end{aligned} \tag{1}$$

The problem amounts to finding the smallest common divisor of N_α^C and N_β^C which is larger than $\sqrt{\frac{N_\alpha^C \cdot N_\beta^C}{M_{tot}^P}}$.

The domain decomposition just discussed creates non-overlapping subdomains. In the ideal scenario whereby all adsorbate species are mono-dentate, and both the energetic interactions and the reaction events are limited to single-site patterns only, the execution of an event in one domain does not affect any events in other domains. In this case, the KMC simulation is trivially parallel and each PE can proceed independently.

However, most surface phenomena in practical applications include elementary events that involve more than one site, for example diffusion of an adsorbate from one site to another. Such events lead to coupling (causal relations) between the simulated histories of neighboring domains. Coupling may also arise from adsorbate-adsorbate lateral interactions and/or multi-dentate species, whereby energetic patterns or adsorbate binding patterns spread over multiple domains. To handle such cases, each PE must store and



(a) PE-1 (b) PE-3

(c) PE-0 (d) PE-2

Figure 2: Domain-sites and halo-sites of processes. For example, in fig. 2c, sites colored in red are the domain-sites and sites in other colors are halo-sites of PE-0.

process information about sites that are close to the boundaries but belong to neighboring domains. The region that contains these additional sites is referred to as halo, and the sites themselves are referred to as halo-sites. The width ω of the halo (in number of sites) is given by

$$\omega = (d_{max} - 1) + (e_{max} - 1) + (r_{max} - 1) = d_{max} + e_{max} + r_{max} - 3 \quad (2)$$

where d_{max} is the maximum denticity among all adsorbate species (the denticity is the number of sites occupied by a species), e_{max} is the maximum depth among all energetic patterns, and r_{max} is the maximum depth among all reaction patterns.

As a simple example, let us consider again the 6×6 periodic square lattice of fig. 1 and a single mono-dentate adsorbate species. In terms of energetic patterns, let us consider only a single body pattern (with energy contribution equal to the adsorption energy of that species); the maximum depth among energetic patterns is one. If single-site adsorption and desorption are the only elementary events considered in the reaction mechanism, the maximum depth among reaction patterns is also one. The width of the halo (eq. (2)) is zero, and the corresponding domain decomposition of the lattice among four PEs is shown in fig. 1. If, however, the diffusion of an adsorbate from its bound-site to a neighboring site is considered as another elementary event, the maximum depth among reaction patterns is two, and the width of the halo becomes one. The corresponding domain decomposition including halo-sites is shown in fig. 2.

As in a sequential KMC run, each PE in the distributed run manages a *procQueue* datastructure that holds all possible KMC events associated with its domain. To avoid duplication of events in the *procQueue* datastructures of different PEs, we impose the following rule: a PE is permitted to add a detected lattice event to its *procQueue* only if the lattice site that is mapped to the first site of the reaction pattern is a domain-site (i.e. it is located in the interior of the domain and not in the halo of that PE).

As an example, let us consider the sites that are handled by PE-1 (fig. 2a), for a system with adsorption, desorption and diffusion events in the mechanism. Considering a state whereby all the sites are empty, the possible events are adsorptions on each and every site of the lattice. However, PE-1 adds the adsorption events only associated with its domain-sites (i.e. the sites that are highlighted in blue in fig. 2a). Adsorption events on halo-sites (represented by a color other than blue in fig. 2a) are not the responsibility of PE-1, as

the lattice site that is mapped to the first (and only, in this case) site in the pattern (adsorption) is a halo-site.

Let us now consider the adsorption on site (11) as the most imminent event and execute it. As site (11) is a domain-site of PE-1 and not a halo-site of any of the neighboring PEs, adsorption on site (11) is an internal-event. In general, we will use the term "internal-event" to refer to a lattice event that involves only domain-sites of one PE and no halo-sites of any PE. Once the adsorption on site (11) is executed, PE-1 deletes this event from its *procQueue*, and adds therein five new events which have now become feasible: diffusion of the newly added adsorbate from site (11) to any of its neighboring sites (5), (10), (12), (17) as well as desorption from site (11).

Continuing with our example, let us assume that diffusion from site (11) to site (5) is currently the most imminent event and execute it. Both sites are domain-sites of PE-1, but site (5) is also a halo-site of PE-3; such events that involve at least one halo-site will henceforth be referred to as boundary-events. Whenever such a boundary-event occurs, it has to be communicated, as a message, from the PE that executes it to the neighboring PEs that share any of the lattice sites involved in the boundary-event. Here, in our case PE-1 communicates the boundary-event to PE-3, as site (5) is in the halo of the latter PE. The message contains all the information about the boundary-event (the KMC time thereof, the sites on which it took place, etc.), so that the message receiver can execute that boundary-event at exactly the same KMC time as the message sender. In our example, when PE-3 receives the message about the diffusion of adsorbate onto site (5), it schedules the event and updates the occupancy of site (5) at the appropriate KMC time.

For the seamless operation of a distributed KMC simulation, one thus requires a mechanism for sending and receiving messages "encoding" lattice events, as well as a protocol for acting on a message that was received. The latter is non-trivial; as noted in the introduction, the asynchronous nature of the distributed KMC simulation may result in cases whereby a messaged event should be scheduled *in the past* of the receiving PE. This leads to a causality violation, since the "history" of the receiving PE has no record of that event, or any subsequent events caused by it. Clearly, corrective actions must be taken to resolve the causality violation and re-simulate a KMC trajectory that takes into account the messaged event. The Time-Warp

algorithm addresses such cases and orchestrates the distributed simulation in a self-consistent way. We will thus continue our discussion with an overview of the Time-Warp concept, followed by our implementation of the different core protocols of the algorithm.

2.3. Outline of the Time-Warp algorithm

The main challenge that the Time-Warp algorithm addresses, is that a causality violation (due to a messaged event that should be scheduled in the past) may have far-reaching effects. This issue will be discussed in more detail later, but as a motivating example consider the diffusion from site (11) to site (5) of fig. 2, which, as discussed in the previous section, has to be communicated from PE-1 to PE-3. Suppose the timestamp of the diffusion event was 1.3 ns while the KMC time of PE-3 was already 2.1 ns when the message about that diffusion was received. Clearly, PE-3 will have to roll-back to a time just before 1.3 ns, discard the history simulated from that time-point onward, schedule the aforementioned diffusion event and simulate any ensuing events.

The problem is that the discarded history may contain events that were messaged to other PEs e.g. an adsorption event on site (28) at time 1.5 ns, communicated to PE-2. Suppose that the latter PE is at KMC time 2.0 ns and we have an effective way of communicating an "undo" action for this adsorption. In turn, while PE-2 is correcting its history from 1.5 to 2.0 ns, it may have to communicate similar "undo" actions to other PEs. In the worst-case scenario, the "original" causality violation (due to the first message encoding a past event) leads to cascade of such violations that require corrective actions at a *global level*, i.e. involving *all* PEs. In practice, however, the effects of a causality violation usually do not extend beyond a certain limited range (dependent on the system simulated).

Thus, the objective of the Time-Warp algorithm is to resolve causality violations by using as much as possible "local" operations, i.e. computing or data-storing operations carried out in private by a PE, or messaging operations that involve only "neighboring" PEs. With these key concepts in mind, we can now discuss the main elements of the Time-Warp algorithm, and the implementation thereof to GT-KMC simulations. Thus, the main data-structures of Time-Warp KMC simulations are:

- *Messages* and *anti-messages* for the communication of boundary-events:

  - A message is an instruction for the receiver to schedule and execute the corresponding event at the appropriate time.
  - An anti-message is an instruction for the receiver to undo the event of a previously received message. A message always precedes the corresponding anti-message.

- Two queues storing information about events, maintained by each PE:
  - A process queue (*procQueue*) that stores information about events that are associated with the domain-sites of the PE.
    * As mentioned before, only events in which the lattice site that is mapped to the first site of the reaction pattern is a domain-site are added to the PE's *procQueue*.
    * Events in which none of the sites involved belongs to the halo of any PE are referred to as internal-events and do not need to be communicated.
    * Events in which at least one halo-site is involved are referred to as boundary-events. A boundary-event needs to be communicated to the neighboring PEs to which the involved halo-sites belong.
  - A message queue (*messgQueue*) that stores information about boundary-events communicated among neighboring PEs as messages. The boundary-events that are received by a PE are referred to as "foreign-events", to distinguish them from the boundary-events that are sent by that PE to others.

- A time-sorted queue of KMC states (*stateQueue*), stored by each PE to support the re-instating of an old KMC state in case of a roll-back due to a causality violation.

The relevant procedures of Time-Warp KMC simulations are summarized as follows:

- In each iteration of the main KMC loop, a PE checks for incoming messages or anti-messages.
  - If a *message* is received with time-stamp greater than the KMC time of the receiver PE (i.e. "encodes" a future event), it is added to the receiver's *messgQueue*.

  - If an *anti-message* with a future action is received, the corresponding message is deleted from the receiver's *messgQueue*.
  - On the other hand, received messages or anti-messages that encode actions in the past are referred to as "stragglers" and cause causality violations. The straggler with the lowest time-stamp is called "priority-straggler". Handling stragglers is done in a similar way to what was discussed above, with the additional corrective actions necessary to restore causality in the KMC history.

- When causality violations occur they have to be handled appropriately:
  - A PE performs a roll-back in time, thereby reinstating a KMC state that has time-stamp less than that of the priority-straggler.
  - During this roll-back, the PE issues anti-messages for all the no longer valid messages (i.e. the previously sent messages that have time-stamp greater than the priority-straggler's time-stamp). Once the anti-messages are sent, the PE deletes the corresponding messages from its *messgQueue*.
  - The PE also deletes any KMC states that have time-stamp greater than or equal to the time-stamp of the priority-straggler, from its *stateQueue*.
  - Finally, the PE re-simulates the KMC history accounting for the newly received message(s) or anti-message(s).

- In each iteration of the main KMC loop, a process executes the first among: (1) the imminent event from its *procQueue* and (2) the imminent foreign-event from its *messgQueue*.

- If the most imminent event is a boundary-event, the PE communicates it as a message, to the neighboring PEs that share any of the sites involved therein. The occurrence time of the boundary-event is communicated as the message's time-stamp.
  - The message being sent is stored in the sender's *messgQueue*, so that if/when needed, in the case of a causality violation, the corresponding anti-message can be sent.

In the next few sections we will discuss these procedures in more detail, starting with the implementation of message handling.

Field name	type
sender	integer
receiver	integer
time-stamp	double
sign	bool
eventtype	integer
nsites	integer
sites(:)	integers
propensity	double
ΔE_{rxn}	double

Table 1: Message data-structure

2.4. Messages and message queue data-structures

Messages in the Time-Warp GT-KMC encode adsorption, desorption, reaction or diffusion events that occur on the lattice, around the boundaries of domains (more precisely, involve at least one halo site). A message is implemented as a data-structure (table 1) with the following fields: *sender* and *receiver* (integers giving the ranks of the corresponding PEs), *timestamp* (double: the KMC time when the lattice event occurred), *sign* (boolean variable, true for a message, false for an anti-message), *eventtype* (an integer giving the type of elementary event, e.g. O_2 adsorption, CO oxidation etc.), *nsites* and *sites* (integers: the count of sites involved in the event and the lattice site numbers thereof), and finally *propensity* and ΔE_{rxn} (doubles: the rate constant and the reaction energy of the lattice event).

Every PE maintains a time-sorted, or more generally priority-sorted, message queue (*messgQueue*) to handle sent and received messages. The message queue is implemented as a doubly linked list [61] in which messages are stored in the nodes. Each node in *messgQueue* also contains a forward and a backward pointer, which point to the queue's next message and previous message, respectively. Insertion of a node/message, deletion of a node/message, and traversal of the queue are the procedures associated with *messgQueue*. The queue also contains a pointer to the first received message that has not yet been executed, which represents the most imminent foreign-event.

2.5. Sending and receiving messages

The preparation of the message to be sent, makes use of a sequence of calls to the **MPI_Pack** subroutine, in order to "pack" the message datatype (table 1) to contiguous memory. To ensure the robustness and efficiency of our implementation, sending and receiving of messages happens in synchronous mode, but with the use of non-blocking MPI subroutines. This way, we have complete control of message buffering as per Algorithms 2 and 3, and at the same time we prevent deadlocks. Thus, the sending of a message starts with the **MPI_Issend** subroutine (line 15 of Algorithm 3), implementing a non-blocking (I) synchronous (s) send. For each initiated communication request, **MPI_Issend** provides a request-handle that can subsequently be used by subroutine **MPI_Test** (line 7 of Algorithm 2) to assert whether the corresponding message has been received. The sending of a message is complete only when **MPI_Test** verifies the receipt thereof. In the meantime (i.e. from the start of the sending with **MPI_Issend** till successful receipt verified by **MPI_Test**), the message has to be retained in a buffer, along with the corresponding MPI communication request-handle.

Algorithm 2: Procedure for verifying receipt of buffered messages and finding first empty slot in message buffer

Data: n_{buffered}: number of buffered messages to test.
MPI_SendReqs(:): 1-d array storing the MPI communication request-handles.
t_{buffered}(:)(double): 1-d array of time-stamps of buffered messages

Result: *lavail_free_slot*(bool): true if there is an available slot in the message buffer, false otherwise
idx_first_free_slot(integer): the first free slot in the buffer (0 if no free slot exists).
updated t_{buffered}(:)

```
1  begin
2      Initialize lavail_free_slot = false;
3      Initialize idx_first_free_slot = 0;
4      for i_message = 1, n_buffered do
5          Set message_sent = false ;
           // If t_buffered(i_message) = ∞, either the slot i_message in
              the buffer has not yet been filled with a
              message or the receipt of a previously buffered
              message in this slot was verified, and the
              time-stamp was reset to ∞.  Otherwise:
6          if t_buffered(i_message) < ∞ then
7              call MPI_Test(MPI_SendReqs(i_message), message_sent) ;
8              if message_sent then
9                  t_buffered(i_message) = ∞ ;
10                 if ¬ lavail_free_slot then
11                     lavail_free_slot = true;
12                     idx_first_free_slot = i_message;
13                 end
14             end
15         end
16     end
17 end
```

Algorithm 3: Procedure for buffering a message and initiating a send request

Data: M(message data-structure of table 1): the message to be sent
n_{buffered} (integer): the number of send-message requests stored in the buffer (up to $n^{max}_{\text{buffered}}$, which is the buffer size).
MPI_SendBuffer(:,:): the buffer (2-d array) of messages
MPI_SendReqs(:): 1-d array storing MPI communication request-handles
t_{buffered}(:)(double): 1-d array containing time-stamps of buffered messages
messgQueue (doubly linked list): priority-sorted queue of messages

Result: updated n_{buffered}, $n^{max}_{\text{buffered}}$, *MPI_SendBuffer*, *MPI_SendReqs*, t_{buffered}, *messgQueue*

1 **begin**
  `// Expression "M.sign" is true for message, false for anti-message`
2 **if** *M.sign* **then**
3 Insert the message in sender PE *messgQueue* ; `// in case a corresponding anti-message needs to be sent later upon a roll-back.`
4 **end**
5 Execute Algorithm 2; `// returns` *lavail_free_slot* `= true if the buffer has free slots and` *idx_first_free_slot* `as the index of the first free slot`
6 **if** $\neg$ *lavail_free_slot* **then**
7 **if** $n_{\text{buffered}} == n^{max}_{\text{buffered}}$ **then**
8 Increase the size of buffer arrays by 100 elements; update $n^{max}_{\text{buffered}}$; `// memory amortization`
9 **end**
10 $n_{\text{buffered}} = n_{\text{buffered}} + 1$;
11 *idx_first_free_slot* $= n_{\text{buffered}}$;
12 **end**
13 Update array element t_{buffered}(*idx_first_free_slot*) with the time-stamp of the message ;
14 Copy message to buffer slot *MPI_SendBuffer*(:,*idx_first_free_slot*) using a sequence of calls to **MPI_Pack** ;
15 Call **MPI_Issend** with buffer-slot *MPI_SendBuffer*(:,*idx_first_free_slot*); store the returned request-handle into *MPI_SendReqs*(*idx_first_free_slot*) ;
16 **end**

To this end, buffered messages are stored in a two-dimensional array, *MPI_SendBuffer*, of size $MessageSize \times n_{\text{buffered}}^{max}$, which is complemented by two additional one-dimensional arrays, *MPI_SendReqs* and t_{buffered} with size $n_{\text{buffered}}^{max}$. The latter variable denotes the maximum number of messages that can be buffered at a given time, while *MPI_SendBuffer* corresponds to the maximum memory that any message would need, which scales with the maximum length of the reaction patterns. Array *MPI_SendReqs* contains the MPI communication request-handles, while t_{buffered} contains the time-stamps of the buffered messages.

The latter array is useful for checking quickly if a buffer slot is empty. Thus, the elements of t_{buffered}, are initialized to ∞, and whenever a message is buffered to $MPI\_SendBuffer(:, i_{\text{message}})$, the corresponding element $t_{\text{buffered}}(i_{\text{message}})$ is updated with the message's time-stamp (line 13 of Algorithm 3). At any later point, if **MPI_Test** returns true for the request-handle $MPI\_SendReqs(i_{\text{message}})$, i.e. successful receipt, the corresponding timestamp $t_{\text{buffered}}(i_{\text{message}})$ is reset to ∞ (lines 6-15 of Algorithm 2). Consequently, at any point in the KMC simulation, $t_{\text{buffered}}(i_{\text{message}}) = \infty$ means that the corresponding elements $MPI\_SendBuffer(:, i_{\text{message}})$ and $MPI\_SendReqs(i_{\text{message}})$ are free to be used for a new communication request. The aforementioned arrays (*MPI_SendBuffer*, *MPI_SendReqs* and t_{buffered}) are memory-amortized, i.e. their sizes can be increased on the fly, as necessary. In our implementation we increase the size of the buffer arrays by 100 elements when they get filled-up (lines 7-9 of Algorithm 3); other strategies are also possible, e.g. increasing the size by a multiplicative factor.

Hence, whenever a PE needs to send a message (or anti-message) to any of its neighboring PEs, it invokes Algorithm 3. First the sign of the message is checked and if this is positive, i.e. the PE is sending a message and not an anti-message, the message is stored in the *messgQueue* (section 2.4) based on its priority (anti-messages on the other hand, annihilate existing messages but are not stored; this is discussed in subsequent sections). Then, Algorithm 2 is invoked, which loops over the first n_{buffered} messages and checks which, if any, have been received, thereby opening a buffer slot. If available, the first of these slots is passed onto Algorithm 3, otherwise slot $n_{\text{buffered}} + 1$ is used (of course Algorithm 3 checks if the buffers are full on line 7, and allocates more space if needed). In the end, the message is copied in the buffer arrays and the send request is initiated with **MPI_Issend** (line 15).

Having discussed the procedures invoked by a sender PE, we now continue to describe what happens on the receiving end. Thus, in every itera-

tion of the KMC loop, Algorithm 4 is invoked, by which PEs first probe for new messages using **MPI_Iprobe** and, if there are any, they receive them with **MPI_Recv** (lines 4 and 6 in Algorithm 4). Note that **MPI_Recv** is a blocking subroutine, which, however, is called *only* if there are new messages, thereby avoiding deadlocks. The received messages are unpacked using a sequence of **MPI_Unpack** calls invoked in line 7 of Algorithm 4 and handled as necessary: messages are stored in *messgQueue*, while anti-messages annihilate existing messages therein. The minimum value among the time-stamps of all received messages is stored in $t^{min}_{\text{received}}$ (line 13 in Algorithm 4; note that $t^{min}_{\text{received}}$ is reset to ∞ at the beginning of Algorithm 4 in line 2).

The value of the latter variable is critical in detecting potential causality violations. Therefore, in an iteration of the KMC loop, there are two possibilities for the relation between KMC time of a process (t_{KMC}) and $t^{min}_{\text{received}}$. The first possibility is $t^{min}_{\text{received}} > t_{\text{KMC}}$, whereby all received messages in the current iteration are in the PE's future. In this case, there is nothing further to be done by the receiver PE at this point. All the received messages are already inserted in *messgQueue* based on their priority. The second possibility is $t^{min}_{\text{received}} \leq t_{\text{KMC}}$, whereby at least one received message in the current iteration is in the PE's past. This constitutes a causality violation. Received messages that are in the past are referred to as *stragglers* [62], and the straggler with the highest priority (lowest time-stamp, $t^{min}_{\text{received}}$) is referred to as the *priority-straggler*.

Resolving a causality violation entails discarding the "wrong" history and restoring an earlier KMC state whose KMC time is before the time-stamp of the priority-straggler. Thus, the simulation effectively "goes back in time". Then, re-simulating the KMC trajectory considering the newly received straggler messages would be relatively straightforward, since these messages are now in the "future". Of course, one has to be careful to send anti-messages as appropriate to other PEs, in order to "undo" messaged events that fall within the discarded history. The pertinent procedures are discussed in detail in the next section.

Algorithm 4: Procedure for probing for and receiving incoming messages

Data: *messgQueue* (doubly linked list): queue containing messages
Result: $t^{min}_{\text{received}}$ (double)
updated *messgQueue*

```
1  begin
2      Initialize t^min_received = ∞; lreceive = true;
3      while lreceive do
4          call MPI_Iprobe(..., lreceive); // check for incoming
               message
5          if ¬ lreceive then exit while loop;
6          call MPI_Recv(...) to receive incoming message (or
            anti-message) M;
7          Retrieve message data with a sequence of calls to
            MPI_Unpack(...);
8          if M.sign then
9              Insert the received message M in the messgQueue;
10         else
11             Find and delete the message corresponding to the received
                anti-message M ;
12         end
           // t_message is the time-stamp of the received message
               (or anti-message)
13         if t_message < t^min_received then t^min_received = t_message ;
14     end
15 end
```

Algorithm 5: Procedure to check if there is a causality violation, and, if so, perform roll-back and prepare/update for rollback-propagation mode.

Data: t_{KMC} (double): current KMC time
$t^{\text{final}}_{\text{KMC}}$ (double): user-specified final KMC time
$t^{min}_{\text{received}}$ (double): minimum value among time-stamps of all the received messages in the current iteration of the KMC loop
rollback-propagation (bool): true if the PE is in rollback-propagation mode, false otherwise
$t^{\text{target}}_{\text{rollback}-\text{propagation}}$ (double): the KMC time just before which the rollback-propagation should terminate
stateQueue (doubly linked list): queue containing stored KMC states

Result: updated t_{KMC}, *rollback-propagation*, $t^{\text{target}}_{\text{rollback}-\text{propagation}}$, and *stateQueue*

```
1  begin
2      if t^min_received < t^final_KMC then
3          if t^min_received ≤ t_KMC then
4              Set rollback-propagation = true ;
5              Set t^target_rollback−propagation = t^min_received;
6              Delete KMC states that have time-stamp greater than or
                 equal to t^min_received ;
7              Reinstate the KMC state from stateQueue that has
                 time-stamp less than and closest to t^min_received ;
8          else
9              if rollback-propagation ∧ t^min_received < t^target_rollback−propagation then
                 t^target_rollback−propagation = t^min_received;
10         end
11     end
12 end
```

2.6. Resolving causality violations

Whenever a causality violation occurs, the KMC timeline of the PE from $t^{min}_{\text{received}}$ (the time-stamp of the priority-straggler) onwards is invalid, as the PE had been unable to take into account the events of stragglers. To resolve the causality violation, the timeline of events has to be re-winded (via a *roll-back* procedure), to a time before $t^{min}_{\text{received}}$, and re-simulated while accounting for the newly received messages. This is achieved by reinstating an old KMC state from *stateQueue* that has time-stamp $t^{\text{reinstate}}_{\text{state}} < t^{min}_{\text{received}}$ (we will refer to this as a "safe" KMC state), and then performing *rollback-propagation* from this new t_{KMC} $(= t^{\text{reinstate}}_{\text{state}})$ to just before $t^{min}_{\text{received}}$. Of course, the PE performing the roll-back has to also send out anti-messages for each and every previously sent message that has time-stamp greater than or equal to $t^{min}_{\text{received}}$. These anti-messages ensure that other PEs make the necessary corrections to their simulated histories, so that a globally consistent simulation is eventually achieved.

2.6.1. State queue and the associated roll-back procedure

To be able to reinstate a safe KMC state when resolving causality violations, each PE maintains a time-sorted, or more generally priority-sorted, state queue (*stateQueue*), into which KMC states are stored at regular intervals, e.g. at every 1000 KMC steps. These intervals are adaptive and not necessarily identical among PEs. Similar to the *messgQueue*, *stateQueue* is also implemented as a doubly linked list. A node in the *stateQueue* holds all the necessary information that is needed to repeat/resume the KMC simulation from that earlier time point. This information includes data-structures such as the lattice state, the process queue, the lists of on-lattice events and lateral interaction patterns, but also the current positions (offsets) of output files (since KMC output will also have to be rewound and re-written.

To treat a causality violation that resulted from a priority-straggler with time-stamp $t^{min}_{\text{received}}$, the PE reinstates the KMC state with time-stamp $t^{\text{reinstate}}_{\text{state}}$ less than but also as close as possible to $t^{min}_{\text{received}}$. After this KMC state has been reinstated, the KMC time is $t_{\text{KMC}} = t^{\text{reinstate}}_{\text{state}}$. The PE subsequently deletes the KMC states that have time-stamps greater than or equal to $t^{min}_{\text{received}}$ in order to free up the associated memory, as these states are no longer valid in view of the causality violation in discussion. During the simulation, it is very likely that the allocated memory for *stateQueue* gets filled up with the saved KMC states, before any invalid states are deleted. In such a scenario, every other element in *stateQueue* is deleted to free up memory for new KMC

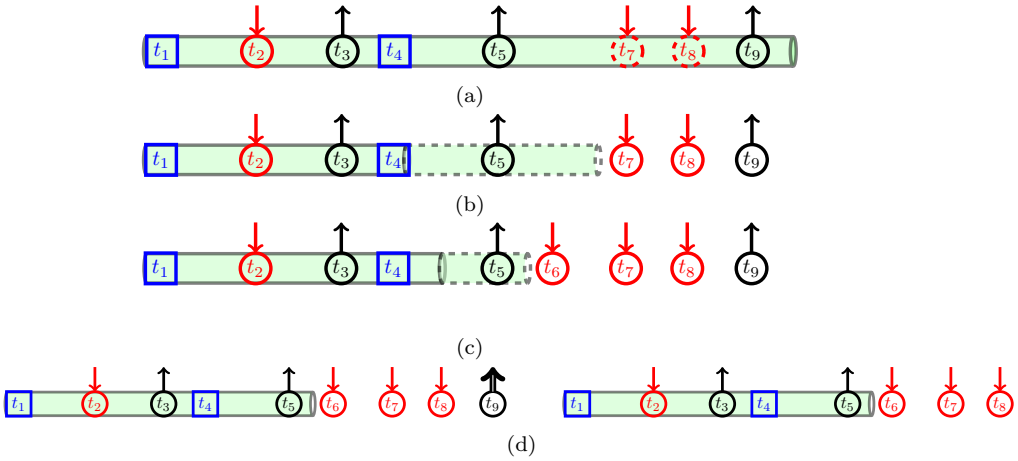


Figure 3: Procedures involved in resolving a causality violation. Pipelines represent KMC time lines. Squares represent KMC state snapshots saved in *stateQueue*. A circle with an inward arrow represents a received message (foreign-event), whereas a circle with an outward arrow represents a sent message (boundary-event). Moreover, a dashed circle with an inward arrow represents a straggler (message with timestamp in the past), while a circle with a double-arrow represents an anti-message. All the messages are stored in the PE's *messgQueue*. Finally, a dashed pipeline represents rollback-propagation. (a) The PE receives two stragglers that have time-stamps t_7 and t_8, and the straggler with the lowest time-stamp (t_7) is the priority-straggler. (b) The PE reinstates the KMC state that has time-stamp $t_4 (< t_7)$, and initiates the rollback-propagation which is supposed to terminate just before executing the priority-straggler (just before t_7). (c) During the rollback-propagation, the PE receives a new message that has time-stamp t_6. As $t_{\mathrm{KMC}} < t_6 < t_7$, the condition for the termination of ongoing rollback-propagation is updated to just before t_6. (d) During the rollback-propagation, the PE does not send messages for the boundary-event with time-stamp t_5, as it was already sent out in the earlier normal-propagation (a). Once the rollback-propagation is finished, just before t_6, the PE sends out anti-messages for each and every previously sent messages that have time-stamp greater than t_6. Here, the PE does this for the message with time-stamp t_9 (left panel) and then deletes this message from its *messgQueue* (right panel).

states (we thus "sparsify" the linked list). This way, there is always a safe KMC state to roll-back to for any possible causality violation.

2.6.2. Rollback-propagation

After reinstating the safe KMC state, the PE enables rollback-propagation mode, aiming at re-simulating the timeline between t_{KMC} and $t^{\mathrm{target}}_{\mathrm{rollback-propagation}}$ (see fig. 3b); $t^{\mathrm{target}}_{\mathrm{rollback-propagation}}$ is the time-stamp of the priority-straggler. During rollback-propagation, the PE executes lattice events (thereby "propagating the KMC state") in almost exactly the same way as under "normal propagation" mode. The only difference is that during rollback-propagation, the PE does not send out messages, even if it executes boundary-events, since the corresponding messages have already been sent out in the first-ever "normal propagation" of the timeline. Rollback-propagation terminates when the PE is about to execute the priority-straggler event, i.e. when $t^{+}_{\mathrm{KMC}} \geq t^{\mathrm{target}}_{\mathrm{rollback-propagation}}$, where t^{+}_{KMC} is the KMC time of the next lattice event. It is important to note that during rollback-propagation, the PE continues to probe for and receive messages in every iteration of the KMC loop. In any iteration of the KMC loop in rollback-propagation, if $t_{\mathrm{KMC}} < t^{min}_{\mathrm{received}} < t^{\mathrm{target}}_{\mathrm{rollback\_propagation}}$, the time before which rollback-propagation will terminate is updated, $t^{\mathrm{target}}_{\mathrm{rollback\_propagation}} = t^{min}_{\mathrm{received}}$ (line 9 in Algorithm 5; see also fig. 3c).

2.6.3. Roll-back of Message queue and anti-messages

As discussed in the introductory paragraph of this section, already-sent messages that have time-stamp greater than or equal to $t^{\mathrm{target}}_{\mathrm{rollback-propagation}}$, are no longer valid and the corresponding events need to be undone. To this end, the PE traverses *messgQueue*, sends anti-messages for each and every previously sent message that has become invalid, and deletes these messages from *messgQueue* (fig. 3d). Communication of anti-messages is similar to the communication of messages; after all, the same data-structure (table 1) is used for both, with the only difference being the value of the "sign" field thereof (true for message, false for anti-message). However, handling these two entities differs: upon receipt of a message, a PE inserts it in its *messgQueue*. On the contrary, upon receipt of an anti-message, a PE finds and deletes the corresponding message from its *messgQueue*. Note that anti-messages can also be stragglers and result in causality violations.

Schematics of the procedures involved in resolving causality violation from the point of view of a single PE and multiple PEs are presented in fig. 3 and

fig. 4, respectively. The pertinent algorithmic elements will be discussed later in the context of the overall distributed KMC algorithm.

2.6.4. Roll-back of KMC output

In addition to rewinding the *stateQueue* and *messgQueue*, one also needs to overwrite the content of output files that is no longer valid, i.e. any KMC output that corresponds to time greater than equal to $t^{\text{target}}_{\text{rollback}-\text{propagation}}$. To achieve this, the output file positions are rewound/rolled-back to the positions stored as part of the KMC state snapshot corresponding to $t^{\text{reinstate}}_{\text{state}}$. Thus, when restoring the earlier "safe" KMC state, the output files are repositioned, so that any new output overrides the invalid old output.

2.7. Global virtual time

As discussed in subsection 2.3, a message encoding a past event may result in a cascade of causality violations, which, in the worst-case scenario, would necessitate corrective actions by all PEs in the distributed simulation. At any given moment during the simulation, the smallest possible time-stamp for such a message is equal to the minimum KMC time among all PEs, assuming of course that no other/previous messages are pending (the more general case will be discussed shortly). Hence, any KMC states generated by the simulation before that time are mutually consistent (thereby following the statistics of the master equation) and will never be discarded during a rollback. This observation leads naturally to the concept of the global virtual time (GVT), which serves as a measure of the progress of distributed KMC runs, similar to the KMC time of a serial run. The GVT is useful as a criterion for terminating the KMC simulation, but also for discarding information that is no longer needed (i.e. KMC state snapshots or messages pertinent to times earlier than the GVT value). Below we will discuss in more detail these two points and their implementation in Time-Warp GT-KMC.

2.7.1. GVT computation and distributed run termination

The computation of the GVT accounts for the KMC times of all PEs but also the timestamps of messages/anti-messages that are in transit i.e. messages that are buffered but not (yet) validated as received (see Algorithm 2 and pertinent discussion in section 2.5). Considering the latter is necessary, since it is not known whether such messages have been acted upon yet, thereby potentially setting the KMC time of a PE to an earlier time. Thus, focusing on a single PE, the minimum among the time-stamps of buffered

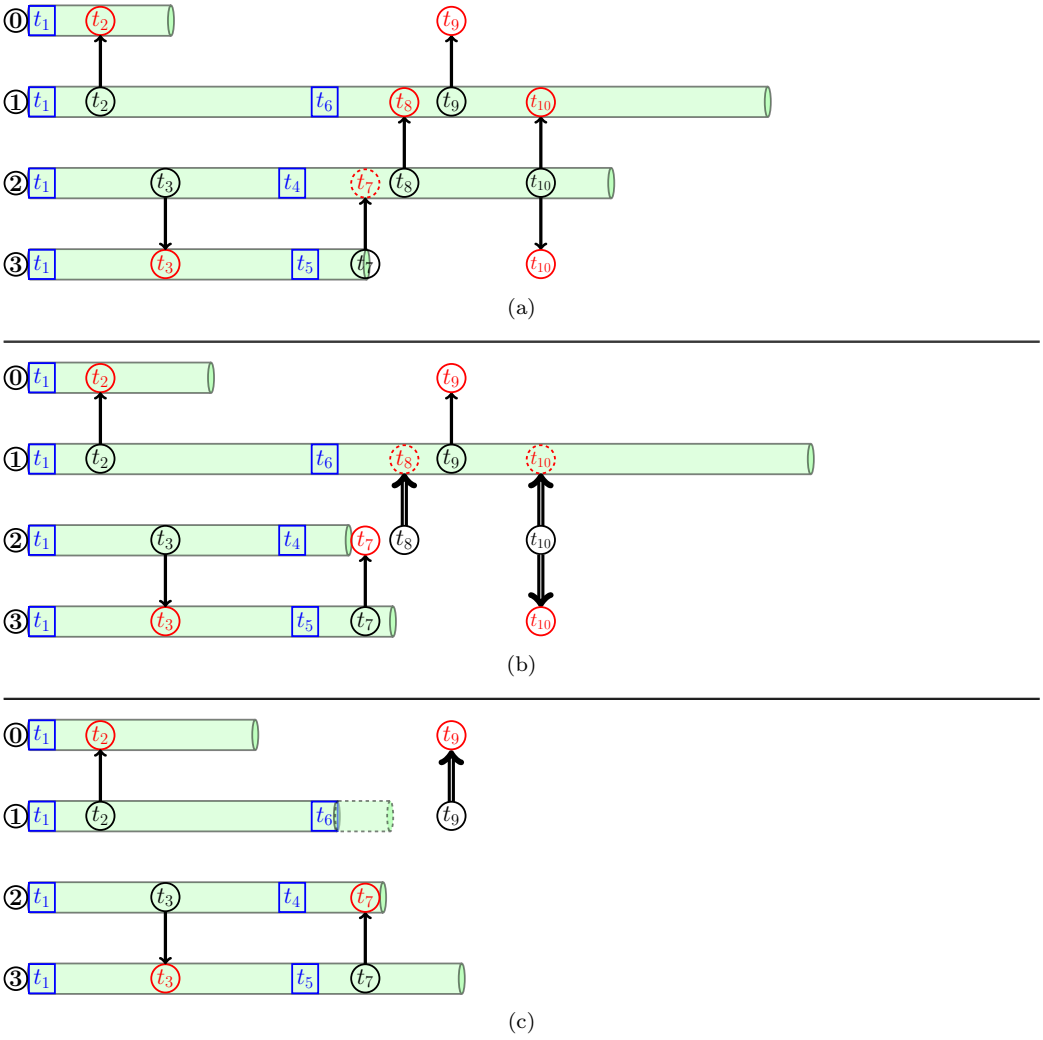


Figure 4: Schematic for different elements in resolving causality violations among multiple PEs. Pipelines represent KMC timelines, and rectangles represent KMC states saved in *stateQueue*. Circles with inward and outward arrows represent received and sent messages, respectively. Circles with double-arrows represent anti-messages. Dashed-circles represent stragglers. (a) PE-2 receives a (priority-)straggler with time-stamp t_7, sent by PE-3, and commits a causality violation, as the message is in its past. (b) PE-2 reinstates the first state (t_4) before the time of the priority-straggler (t_7), and performs rollback propagation that terminates just before t_7. The PE(-2) also issues anti-messages to the previously sent messages that have lower priority than the priority-straggler; messages with time-stamps t_8 and t_{10}. (c) As PE-1 received anti-messages (that have time-stamps t_8 and t_{10}) as stragglers in (b), it reinstates the KMC state that has time-stamp t_6 and initiates rollback-propagation that terminates just before t_8 (time-stamp of the priority-straggler). Note that PE-1 issues the necessary anti-messages (here for the message that has time-stamp t_9) after completing the rollback-propagation.

messages $t_{\rm buffered}(:)$ is evaluated and stored in $t^{min}_{\rm buffered}$. The two variables just noted are local to a PE. The minimum among $t_{\rm KMC}$ and $t^{min}_{\rm buffered}$ is defined as the local virtual time (LVT) (line 7 in Algorithm 6). In turn, the GVT is defined as the minimum among the LVTs of each and every PE in the distributed run. Computing the GVT is thus a global operation and is done at regular, pre-chosen, time intervals by invoking the **MPI_AllReduce** function (line 8 in Algorithm 6).

Algorithm 6: GVT (Global virtual time), "fossil collection" and termination status of the distributed run

Data: $\tau_{\rm prev}$ (double): clock time when the last GVT computation was performed. At the beginning of the simulation, $\tau_{\rm prev}$ is initialized to zero.
$\Delta\tau_{\rm GVT}$ (double): clock time interval for GVT computation, as prescribed in the input.

Result: GVT (double), updated $\tau_{\rm prev}$ (double) and $t_{\rm buffered}(:)$ (1-d array of doubles)

1 **begin**
2 Call system clock function to obtain $\tau_{\rm curr}$, the current clock time ;
3 Set $\Delta\tau = \tau_{\rm curr} - \tau_{\rm prev}$;
4 **if** $\Delta\tau \geq \Delta\tau_{\rm GVT}$ **then**
5 Invoke Algorithm 2 to update the status of buffered messages;
6 Set $t^{min}_{\rm buffered} = min(t_{\rm buffered}(:))$;
7 Set $LVT = min(t_{\rm KMC}, t^{min}_{\rm buffered})$;
8 Perform a global communication by invoking **MPI_AllReduce**(LVT) to obtain $GVT = min(LVT)$;
9 Set $\tau_{\rm prev} = \tau_{\rm curr}$;
10 **if** $GVT \geq t^{\rm final}_{\rm KMC}$ **then** exit main KMC loop and terminate distributed run;
11 Delete obsolete KMC states from *stateQueue*;
12 Delete obsolete messages from *messgQueue*;
13 **end**
14 **end**

As mentioned earlier, calculating the GVT is useful for terminating the distributed KMC run. In general, terminating distributed simulations can be complicated [63], contrary to sequential runs. Thus, in a sequential KMC run, the termination criterion is straightforward: reaching the specified maximum KMC time marks the end of the simulation. In a distributed run, termination detection must be a global process similarly to how the GVT is computed. Thus, given the desired final KMC time, $t_{\mathrm{KMC}}^{\mathrm{final}}$, the distributed run is terminated when GVT is greater than $t_{\mathrm{KMC}}^{\mathrm{final}}$ (line 10 in Algorithm 6). In a distributed run, it may happen that one or more PEs reach the specified maximum KMC time and appear to have terminated locally. However, other PEs may have not finished yet, and therefore the run should not be terminated globally. In this case, the PEs that have terminated locally continue iterating within the main KMC loop but only for the purposes of probing for and receiving any messages, and also making sure all the buffered messages are sent (line 9 in Algorithm 7). It is critical that these PEs do not exit the main KMC loop, because it is certainly possible that they will receive a message with time-stamp less than $t_{\mathrm{KMC}}^{\mathrm{final}}$, thereby committing a causality violation and having to roll-back.

2.7.2. Recovery of memory by deleting obsolete KMC states and messages

Since the GVT is the minimum of the LVTs of all PEs, it is guaranteed that a priority-straggler (message or anti-message) will not have time-stamp strictly less than GVT (note that it is still possible for the time-stamp to be equal to GVT and lead to a causality violation). Consequently, in *stateQueue* one needs to retain only the last KMC state before the GVT, whose timestamp will be denoted with $t_{\mathrm{state}}^{\mathrm{GVT}-}$, and of course all subsequent states. On the other hand, any state with time-stamp less than $t_{\mathrm{state}}^{\mathrm{GVT}-}$ can be safely discarded to free up the associated memory (line 11 in Algorithm 6).

Similarly, no longer needed messages can be deleted as well, to free up memory (line 12 in Algorithm 6). Such obsolete messages have time-stamp less than $t_{\mathrm{state}}^{\mathrm{GVT}-}$, since, if the last KMC state just before GVT is restored, one will need all messages received from $t_{\mathrm{state}}^{\mathrm{GVT}-}$ onward, to correctly resimulate the timeline upon rollback-propagation. The process of recovering memory by deleting the obsolete KMC states and messages is referred to as "*fossil collection*" [62].

Algorithm 7: Distributed KMC algorithm (executed by each PE)

Data: Lattice, energetic clusters/figures, reaction mechanism, and simulation conditions, such as temperature, pressure and gas phase mole-fractions.
Result: A trajectory of lattice states.

```
1  begin
2      Initialize GlobClusterEnerList and add all domain instances of energetic
         patterns to the list;
3      Initialize procQueue with all possible domain processes along with their
         random occurrence times;
4      Initialize rollback-propagation = false ;
5      while True do
6          Invoke Algorithm 4 to probe for and receive incoming messages;
7          Invoke Algorithm 5 to check for causality violation and act accordingly,
             or update the information for rollback-propagation, if necessary ;
8          Invoke Algorithm 6 to compute GVT and decide on run termination ;
9          if t_KMC ≥ t_KMC^final then cycle ;
10         if rollback-propagation ∧ t_KMC^+ ≥ t_rollback−propagation^target then
11             Set rollback-propagation = false ;
12             Issue anti-messages for each and every previously sent message that
                 has time-stamp greater than or equal to t_rollback−propagation^target ;
13         end
14         Obtain the most imminent event E_next among procQueue and
             messgQueue;
15         if ¬ rollback-propagation then
16             if E_next is foreign-event and is not feasible then cycle;
17             Set boundary-event = false ;
18             if E_next is from procQueue and involves at least one halo-site then
                 set boundary-event = true ;
19             Store a KMC state snapshot in stateQueue, if timely ;
20         end
21         Execute the most imminent event;
22         if ¬ rollback-propagation ∧ boundary-event then invoke Algorithm 3
             for every PE that has a halo-site affected ;
23         Execute steps 7-21 of Algorithm 1 ; // update operations for the
               assigned domain as per the sequential KMC algorithm
24     end
25 end
```

Due the requirement to save simulation states, the Time-Warp algorithm can be quite memory intensive. Thus, freeing up memory after every GVT computation block is critical. Note, however, that, in our implementation, GVT computation is achieved through global communication, which can limit the performance if invoked very frequently, especially if the number of PEs is large (for alternative approaches see e.g. Mattern [64]). The choice of the frequency of GVT computation, evidently, is an optimization problem and further investigations on this topic will be part of our future work. In the current implementation, a clock time interval is prescribed as part of the simulation input and GVT computation is invoked at regular such intervals.

2.8. Distributed KMC algorithm

The distributed KMC driver is summarized in Algorithm 7, which has been implemented in *Zacros* and makes use of the Time-Warp algorithm protocols previously discussed. An additional important technical point pertains to the feasibility check for the most imminent event (line 16). In a sequential run, the PE never encounters a situation where an imminent event is infeasible. However, in a distributed run, it is possible that the most imminent event is a foreign-event E_2 that could depend on another foreign-event E_1, but the corresponding message of E_1 may have not yet been received by the PE. In this case, the most imminent event (E_2) is infeasible and prevents the PE from progressing forward. It is important to note that this is only a temporary setback (the Time-Warp algorithm by conception prevents deadlocks [43]). To address it, the PE cycles the KMC loop keeping the KMC state "as is" until the infeasibility is resolved (line 16 in Algorithm 7). The situation gets resolved in two possible ways, either by receiving a message for the foreign-event E_1 which has higher priority than the infeasible foreign-event, or by receiving an anti-message for the infeasible foreign-event (E_2).

As an example, let us consider the following scenario to demonstrate how event infeasibility may arise. The scenario involves events on sites (15) and (21), which are shared by all four PEs (see fig. 2). Initially, site (21) is occupied with a mono-dentate adsorbate and all other sites, including (15), are empty. PE-2 executes a diffusional hopping of the adsorbate from site (21) to site (15), and, as this is a boundary-event, communicates the event as a message to each of the other PEs. The order by which the other PEs receive these messages is unknown. It is thus possible that PE-0 receives the

corresponding message first and executes a consequent event, desorption of the adsorbate from site (15). Clearly, PE-0 has to communicate the desorption event just executed to all other PEs since site (15) is in their halos. It is now possible that PE-1 receives the message about the desorption from PE-0 before receiving the message about the diffusion from PE-2. In this case, the foreign-event that PE-1 would try to execute next (the desorption) is infeasible, as there is no adsorbate on site (15) yet. PE-1 will have to continue iterating the KMC loop until it receives the message about the diffusion from PE-2, so as to resolve the infeasibility problem.

2.9. Parallel-emulation

Implementing the Time-Warp algorithm to enable distributed parallelization of GT-KM is not trivial and, thus, validation is essential to ensure correctness. Such a validation approach can be based on the following constraint inherent to the Time-Warp algorithm: "if event A has higher priority[1] relative to event B, event A should be scheduled and executed before event B" ("before" in this context is meant in real/clock time terms). This constraint can be used to devise a "parallel-emulation" scheme, by which sequential (serial) KMC runs generate output identical to that of distributed runs, without the use any of the Time-Warp algorithm protocols. Validating the correctness of distributed runs can then be done in a straightforward way by comparing their results with those obtained from the corresponding parallel-emulation runs.

Thus, a "parallel-emulation" run is a serial KMC run which uses exactly the same random numbers as the distributed run, and of course, in the same order. To achieve this, the following algorithmic elements are needed: (1) a set of N_P random stream instances (objects) that are identical to those used by the N_P PEs in the distributed run (as one stream is used for each domain); (2) a function that returns the domain to which a lattice event

[1]In our discussion, priority is determined on the basis of the timestamp of an event. It is possible (though quite infrequent in practice) that two events have equal timestamps, in which case, additional priority rules have to be imposed in order to ensure the seamless operation of the Time-Warp implementation. These rules are discussed in detail in the Supplementary Information and are adopted in our Time-Warp implementation. It is possible to also implement them in the parallel-emulation scheme; however, given that equal-timestamp events are rare, we chose to keep the latter as simple as possible.

"belongs". This "domain of ownership" is unambiguously determined, simply by inspecting the lattice site onto which the first site of the reaction pattern has been mapped. Recall that in a parallel run, an MPI process adds a detected lattice event to its *ProcQueue* only if the site 1 of the pattern is mapped to a domain-site. This rule can be easily embedded into the function in discussion, without the need for the serial run to actually perform the domain decomposition.

Within this framework of the parallel-emulation run, whenever a new event is identified, the algorithm first determines the domain to which the event belongs and then uses the corresponding random number stream when generating the KMC time of occurrence for that event. Additionally, for simulations with lateral interactions between adsorbates, the correct random number has to be chosen whenever a time of occurrence is updated (recall that in GT-KMC such updates are needed for any event affected by changes in the neighboring spectators due to the lattice event just executed [37]). Thus, a parallel-emulation run is essentially a serial run, with a slightly modified procedure for random number generation. In such a run, there are no communications, causality violations, or roll-backs, and none of the Time-Warp specific data-structures (e.g. *messgQueue*, *stateQueue*), or procedures (e.g. GVT computation) are used.

Of course, such runs are expected to be quite intensive for large lattices, since data and workload are no longer distributed among several PEs. In addition, parallel-emulation runs are slightly less efficient than "traditional" serial runs, due to small memory overheads in maintaining several random streams and computational overheads in selecting the appropriate random number out of these. Hence, the parallel-emulation scheme is only used for validating distributed runs by eliminating the underlying complexity of Time-Warp.

3. Results and Discussion

We proceed to discuss the validation and performance benchmarks of our implementation of the Time-Warp GT-KMC approach. The latter has been incorporated into *Zacros* [18, 37, 38], our FORTRAN 2003 code for KMC simulations in catalysis and surface science. For validation and benchmarking we have employed two simple models (systems 1, 2), both simulated on square lattices with one site per unit cell (similar to those of fig. 1 and fig. 2), while for demonstrating our implementation on a more realistic system, we have

$k_{\text{ads/des}}$ (s^{-1})	k_{diff} (s^{-1})	$\varepsilon_{\text{site}}$ (eV)	$\varepsilon_{\text{CO}^*}$ (eV)
1.0	10.0	0.0	0.1

Table 2: Rate constants of elementary events, CO^* adsorption, CO^* desorption and CO^* diffusion. Interaction of CO^* with the bound-site ($\varepsilon_{\text{site}}$) and lateral interaction between adsorbed CO^* molecules (ε_{CO^*}) are also given. Note that CO^* diffusion is exclusive to system 1 and lateral CO^*-CO^* interactions are exclusive to system 2. Simulations are run at a temperature of 500 K and pressure of 1 bar.

chosen a detailed model of CO oxidation on Pd(111) (system 3, taken from Ref. [65]). A brief description of the main features of these systems is provided below.

- **System 1** entails CO adsorption, CO^* desorption, and CO^* diffusion, without lateral interactions.
- **System 2** entails CO adsorption and CO^* desorption, with nearest-neighbor lateral interactions among CO^*.

All events in these two systems are taken as reversible and can be summarized by the following "reactions", in which $*$ refers to a vacant site, subscript (g) denotes a gas species, and superscript $*$ denotes a surface species:

$$\mathrm{CO_{(g)}} + * \underset{k_{\text{des}}}{\overset{k_{\text{ads}}}{\rightleftharpoons}} \mathrm{CO^*} \tag{3}$$

$$\mathrm{CO^*} + * \underset{k_{\text{diff}}}{\overset{k_{\text{diff}}}{\rightleftharpoons}} * + \mathrm{CO^*} \tag{4}$$

The aforementioned two systems were chosen such that inter-domain coupling arises from different factors. In system 1, adsorption/desorption and diffusion are considered in an ideal adlayer (no lateral interactions), and domain dynamics are coupled via diffusion, which results in particles crossing domain-boundaries. On the other hand, system 2 involves only single site events (adsorption/desorption), but entails coupling due to pairwise lateral interactions, which may result in a particle of one domain affecting the desorption rate of a particle in another domain. The values of the parameters used in the benchmark simulations are given in table 2.

- **System 3** entails 22 elementary events capturing CO oxidation dynamics on a Pd(111) lattice with two site types (fcc and hcp). The lattice size was kept fixed to $2{,}592 \times 2{,}592$ unit cells with two sites each (13,436,928 sites in total).

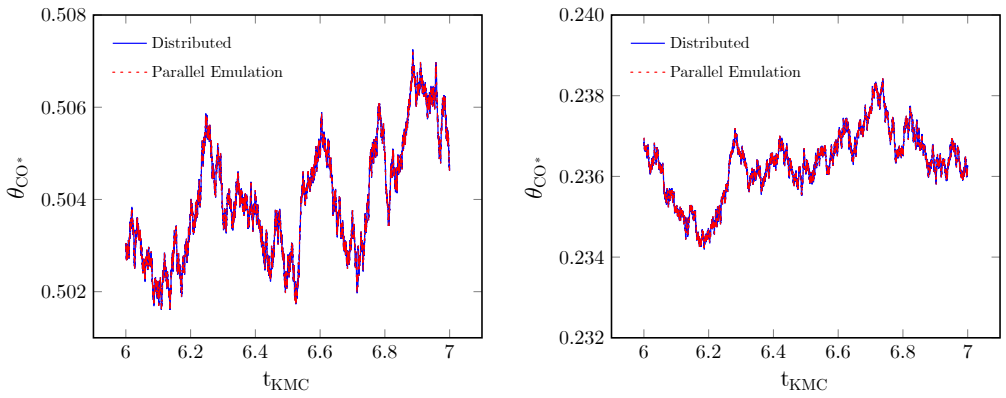
More details on this system can be found in our previous publications [65, 66, 67]. Briefly, the elementary events include the adsorption, desorption and diffusion of CO and O_2, the dissociation of O_2 and the diffusion of the resulting O^* adatoms, as well as the reaction of CO^* with O^* towards CO_2. All events include variants as appropriate, as they can occur on the two different site types. Lateral interactions are captured by a cluster expansion with 88 patterns, including single-, two- and three-body patterns. The maximum reaction pattern "depth" is 3 sites (for O_2 dissociation on an fcc site, which gives rise to two O^* on hcp sites; a similar event exists for O_2 dissociation on an hcp site). Moreover, the maximum energetic interaction pattern "depth" is also 3 sites, due to the inclusion of 3-body patterns. These depths and the connectivity of the sites result in a halo width of 4 unit cells.

The parameters of the cluster expansion are as of model 2 of fig. 8 in Ref. [65], for a temperature of 440 K and a pressure of 10^{-6} bar (fourth point from the left in the blue curve of the figure just noted). This model was one of the variants developed as part of a sensitivity analysis study in Ref. [65] and incorporates: adsorption stabilization for CO of 0.2 eV as suggested by experiments and RPA calculations (i.e. $E_{ads} = -1.5276$ eV on the fcc site and -1.5047 eV on the hcp), adsorption stabilization for O_2 of 0.3 eV (i.e. $E_{ads} = -0.9712$ eV on the fcc site and -0.8256 on the hcp site), and an activation energy for O_2 dissociation lowered by 0.2 eV (i.e. 0.59 eV and 0.45 eV, for dissociation on fcc and hcp sites, respectively).

3.1. Validation of the Time-Warp algorithm implementation

Our Time-Warp GT-KMC implementation is validated against parallel-emulation runs, as described in section 2.9. Thus, a correct implementation of the Time-Warp algorithm is expected to deliver KMC trajectories that are in *exact agreement* (down to the stochastic fluctuations) with the results from the corresponding parallel-emulation runs. The results of the validation runs are shown in fig. 5(a) for system 1 and fig. 5(b) for system 2.

The corresponding Time-Warp and parallel-emulation runs take place on 100×100 lattices, initialized with all sites vacant. For each such run, we plot the coverage of CO* over time for t_{KMC} ranging from 6 to 7 units of KMC time (simulated seconds), so that we focus on the fluctuations. Clearly, for both systems the distributed simulations produce identical results to those obtained from the parallel-emulation scheme, thereby validating our implementation.



(a) System 1

(b) System 2

Figure 5: CO* coverage profiles over time obtained from distributed runs that use the Time-Warp algorithm versus the corresponding parallel-emulation runs. The coverage is shown between KMC time of 6 and 7 simulated seconds to highlight the fluctuations. Panels (a) and (b) show the results from the distributed and parallel-emulation runs for systems 1 and 2, respectively. The identical profiles obtained for each system provide evidence for the correctness of the Time-Warp GT-KMC implementation.

3.2. Performance benchmarks

Having validated our implementation, we now investigate the performance thereof, in terms of both weak- and strong-scaling. All performance benchmarks of systems 1 and 2 were carried out on the high performance computing cluster Kathleen at UCL (`https://www.rc.ucl.ac.uk/docs/Clusters/Kathleen/`), while for system 3, we made use of cluster Thomas hosted by the Materials and Molecular Modelling (MMM) Hub (`https://www.rc.ucl.ac.uk/docs/Clusters/Thomas/`). As a raw metric of performance we use the KMC-time advancement for a fixed real/clock-time interval. It is therefore important to perform the simulations under stationary conditions, under which the average number of KMC events per unit of KMC-time per lattice area remains constant. In this case, KMC-time advances (on average) linearly with respect to clock time, and, in the ideal case of a trivially parallel KMC simulation (without halos), the pertinent "speed" is inversely proportional to the number of sites in the lattice. These statements will be expressed in quantitative terms later, when we define efficiency factors; at this point our intention is to highlight the procedures adopted to eliminate any bias in our investigations. Hence, in order to initialize our simulations from lattice configurations representative of the system's behavior in stationary conditions, a long KMC simulation is first run on a small size lattice until stationary state is reached. The final state of the lattice is then tessellated (tiled) as needed to generate initial state input for a larger lattice, which is subsequently provided as the input state for the scaling benchmark simulations.

There are three input parameters that allow us to tune the performance of the distributed runs: the memory (per PE) available for saving KMC states in *stateQueue*, the frequency of saving KMC states, and the clock-time interval for GVT computation. Their values were chosen as shown in table 3, in order to optimise performance while respecting the constraints of the different systems and PE configurations. For example, for system 3 only strong scaling benchmarks were possible due to the large lattice size and long simulation times.

Thus, in both weak and strong scaling benchmarks, the memory allocated exclusively for *stateQueue* is set to 4 GB for system 1, and 3.5 GB for system 2. For system 3, we performed only strong scaling benchmarks. Due to the size of the lattice used in system 3 ($2{,}592 \times 2{,}592$ unit cells, each with two sites), the per-PE memory dedicated to *stateQueue* was more on configurations with a small number of PEs. More specifically, we used 15 GB for the

System	Scaling benchmark	MPI configuration[(i)]	*stateQueue* memory (GB)	State-saving interval[(ii)] (#events)	GVT comp. interval (s)
1	Weak	1×1	4.0	10^9 [(iii)]	5
		$2 \times 2 - 30 \times 30$	4.0	100	5
1	Strong	$2 \times 2 - 20 \times 20$	4.0	100	5
2	Weak	1×1	3.5	10^9 [(iii)]	3
		$2 \times 2 - 30 \times 30$	3.5	50	3
2	Strong	$2 \times 2 - 20 \times 20$	3.5	50	3
3	Strong	2×2	15.0	5	10
		3×3	8.0	5	10
		4×4	5.5	5	10
		$6 \times 6 - 27 \times 27$	4.5	5	10

Table 3: Settings of the Time-Warp based runs. Notes: (i) for strong scaling benchmarks, the single-PE runs were serial (hence the absence of 1×1 MPI configurations). (ii) the *initial* state-saving interval is reported. When the state queue gets completely filled, the queue is sparsified and the state-saving interval is doubled (see section 2.6.1). (iii) this setting effectively switches off snapshot saving, which is unnecessary for single-PE runs.

2×2 PE configuration, 8 GB for the 3×3, 5.5 GB for the 4×4 and 4.5 GB for the rest of the PE configurations. Moreover, the GVT computation interval is set to 5 seconds for system 1, 3 seconds for system 2, and 10 seconds for system 3. Regarding the frequency of saving KMC states, we start with saving a state every 100 KMC steps in system 1, 50 steps in system 2, and 5 steps in system 3. If *stateQueue* gets filled up before purging no-longer-needed or invalid states, the queue is sparsified (see section 2.6.1)and the frequency is halved, i.e. KMC states are saved every 200 KMC steps in system 1. This adaptive procedure happens whenever needed in the distributed runs.

Note that these settings were chosen after a small number of (manual) trials towards optimising the performance. A detailed assessment of the effect of these parameters on performance is ongoing work, but out of scope for this paper. Here, we aim at conducting proof-of-concept simulations, to investigate the feasibility of Time-Warp GT-KMC for catalytic kinetics studies.

In our discussion, we use the following definitions and notation. The

scaled time (t^*) is defined as:

$$t^* = \frac{t_{\mathrm{KMC}}}{t_{\mathrm{Clock}}} \tag{5}$$

where t_{KMC} is the elapsed KMC time in a simulation that is run for t_{Clock} clock time (this is what we referred to as the "speed" of the simulation in the previous paragraph). $t^*\left(n_{\mathrm{sites}} : n_{\mathrm{pe}}\right)$ is the scaled time from a distributed run in which a lattice containing n_{sites} sites is distributed over n_{pe} PEs. The scaled time obtained from serial runs is represented as $t^*\left(n_{\mathrm{sites}}\right)$. Note that $t^*\left(n_{\mathrm{sites}} : 1\right)$ represents the scaled time from a distributed run that uses only one PE; this may include overheads compared to $t^*\left(n_{\mathrm{sites}}\right)$. The timing data obtained from our benchmark simulations (discussed below) is provided in the supplementary information.

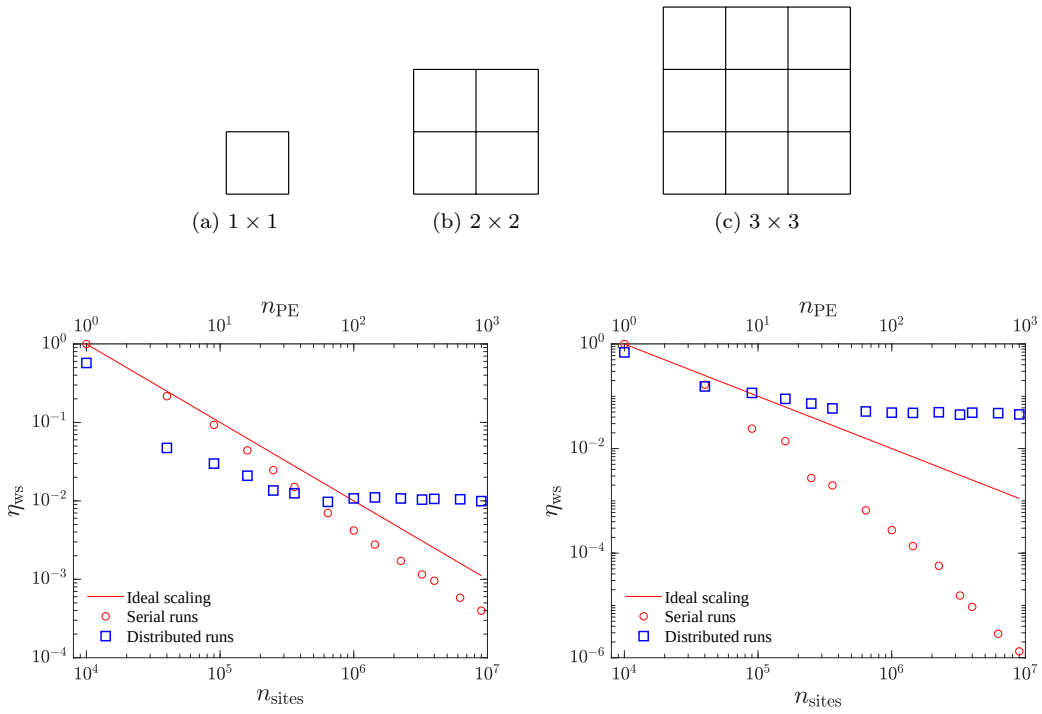
3.2.1. Scaling behavior of serial runs

Before discussing the performance/efficiency of distributed runs that use the Time-Warp algorithm, it is instructive to assess the performance of the serial implementation, which is used as a basis for comparison. To this end, serial runs are carried out for progressively larger lattices, and the measured performance is compared against the ideal scaling law, whereby, in the absence of any overheads, the scaled time t^* would be inversely proportional to the number of sites in the lattice.

The pertinent results are shown in fig. 6d and fig. 6e. The red circles show the efficiency of each serial run, defined as

$$\eta = \frac{t^*\left(n_{\mathrm{sites}}\right)}{t^*\left(n_{\mathrm{sites}}^{min}\right)} \tag{6}$$

with n_{sites}^{min} the number of sites in serial run with the smallest lattice (10,000 sites). The red lines highlight the ideal scaling, whereby $\eta = n_{\mathrm{sites}}^{min}/n_{\mathrm{sites}}$. For system 1 (fig. 6(d)), the performance follows the ideal scaling law for low to moderately sized lattices containing up to $\sim 2 \times 10^5$ sites; for lattices larger than that, the performance starts to deteriorate. For system 2, the serial run performance deviates from the ideal scaling already with lattices that contain $\sim 4 \times 10^4$ sites. The worse performance of the serial algorithm for system 2, compared to system 1, can be attributed to additional operations necessary to compute the rate constants of events in the presence of lateral interactions. More specifically, computing rate constants for desorption events necessitate



(a) 1×1 (b) 2×2 (c) 3×3

(d) System 1 (e) System 2

Figure 6: (a-c): In the weak-scaling analysis the number of PEs is increased proportionally to the lattice size, so that the workload per PE remains constant. (d, e): Weak-scaling performance of the Time-Warp algorithm in *Zacros* for the two benchmark systems. For the weak-scaling analysis, efficiency is calculated using eq. (7). Each subdomain assigned to a PE contains $10,000$ sites.

the detection of lateral interaction patterns in the neighborhood of the event. These detection operations entail solving the subgraph isomorphism problem [18, 37] which introduces noticeable overheads for large systems (especially since GT-KMC simulations need to frequently update the rate constants of existing events in the presence lateral interactions).

3.2.2. Weak-scaling behavior

We continue our discussion with the weak-scaling benchmarks, in which n_{sites} and n_{PE} are both increased proportionately, thereby maintaining a constant workload per MPI-process (recall that the workload scales linearly with the number of domain-sites, in the absence of computational overheads). For

these benchmarks we define the efficiency (η_{ws}) as follows:

$$\eta_{\text{ws}} = \frac{t^* \left(n_{\text{sites}} : n_{\text{PE}}\right)}{t^* \left(n_{\text{sites}}^{min}\right)} \tag{7}$$

with the total number of lattice sites being $n_{\text{sites}} = n_{\text{sites}}^{min} \times n_{\text{PE}}$ as portrayed in fig. 6a - 6c, and as before, n_{sites}^{min} denotes the number of lattice sites in the serial run with the smallest lattice. In the ideal case (trivially parallel problem), the efficiency would be equal to one, since $t^* \left(n_{\text{sites}} : n_{\text{PE}}\right) = t^* \left(n_{\text{sites}}^{min} \times n_{\text{PE}} : n_{\text{PE}}\right) = t^* \left(n_{\text{sites}}^{min} : 1\right)$.

The results of our weak scaling benchmarks are shown in fig. 6d and fig. 6e. Let us first compare the single-PE distributed run with serial run (leftmost points for 10^4 sites in the plots just noted). In the distributed runs with just a single PE, none of the following occurs: point-to-point communications via messages, causality violations, rollback propagation. We have therefore suppressed the handling of *stateQueue* by choosing a very large interval for state saving (every 1 billion events). However, the algorithm still checks at every KMC iteration whether an event involves halo sites (even though there are none in this case). In addition, MPI subroutines that e.g. probe for messages (even though there wouldn't be any), or broadcast the LVT (to just one MPI process) are still called as usual. The pertinent overheads can be assessed by calculating the quantity: $t^* \left(n_{\text{sites}}^{min} : 1\right) / t^* \left(n_{\text{sites}}^{min}\right)$, and our results reveal that such overheads result in a $30 - 40\%$ slowdown when comparing truly-serial runs with single-PE distributed ones (see fig. 6). Of course, simulations of the latter type are not useful in practice and adopting them for production runs would be ill-advised. It is relatively straightforward to improve our implementation by skipping the aforementioned operations when only one PE is used; still though, single-PE runs like the ones we just discussed, are quite useful for benchmarking purposes, as they can reveal time-consuming operations.

The smallest actually distributed simulations are performed for 4 PEs (40,000 sites), as shown in fig. 6d and fig. 6e, for systems 1 and 2, respectively. The observed drop in efficiency for these relative to the respective serial runs for 40,000 sites can be attributed mainly to the emergence of communications, the handling of *messgQueue* and *stateQueue*, and the treatment of causality violations via rollbacks. The latter are indeed quite costly, as they entail discarding "chunks" of KMC trajectories, which were obtained by consuming computational resources. It is however encouraging to see that the performance stabilizes for larger lattices (fig. 6), for which the serial runs are

becoming progressively more inefficient. For the largest lattices investigated, we see that the distributed runs are more than 10 times faster for system 1 compared to serial runs. For system 2, the difference is much higher, with the distributed runs being more than four orders of magnitude faster than serial ones. The better performance observed for this system could be attributed to the fact that updates to reaction rates due to lateral interactions are done as part of the execution of any event, and are decoupled from any Time-Warp related operations. Thus, they are parallelized quite effectively via the domain decomposition scheme of our implementation. These results highlight the potential of the Time-Warp GT-KMC implementation to be used in studies of large lattices, which exceed current capabilities.

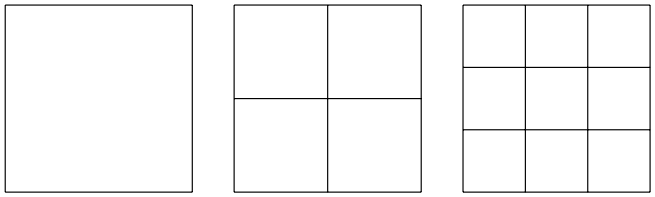
3.3. Strong-scaling behavior

Contrary to the weak-scaling benchmarks, in which the size of the problem increases in proportion to the computational resources, when analysing strong-scaling, the size of the problem remains fixed. Thus, to assess the strong-scaling performance of our Time-Warp GT-KMC implementation we increase n_{PE} for a fixed value of n_{sites} (see fig. 7a-7c), thereby, decreasing the workload per PE. For the strong-scaling benchmarks, the efficiency (η_{ss}) is defined with respect to a serial run, as follows:

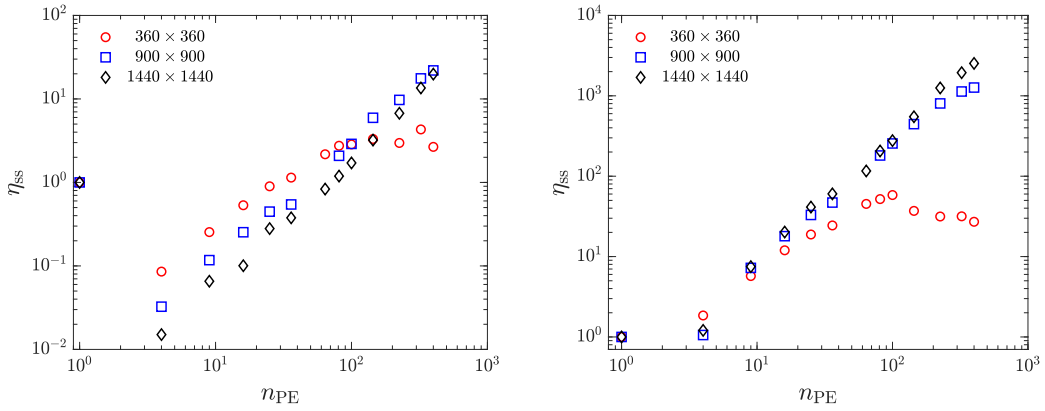
$$\eta_{\mathrm{ss}} = \frac{t^* \left(n_{\mathrm{sites}} : n_{\mathrm{PE}}\right)}{t^* \left(n_{\mathrm{sites}}\right)} \tag{8}$$

The results of these benchmarks for systems 1 and 2, are displayed in fig. 7. For actually distributed runs (i.e. with more than just one PE), we generally observe an improvement in performance as more PEs (n_{PE}) are employed in our simulations. The number of PEs for which a distributed run starts to outperform the single-PE run depends on the system and the lattice size (having fixed the allocated memory for the *stateQueue*, the frequency of state-saving, and the specified time interval for GVT computation). For example, distributed runs with more than 100 PEs outperform single-PE run for all lattices of system 1, while for system 2, runs with 9 PEs already outperform the single-PE runs quite significantly.

In our strong scaling benchmarks, we also observe that the efficiency plateaus for the smallest lattice considered (360×360). This plateau is due to the increasing portion of halo sites. Indeed, as the number of PEs increases, the overall length of boundaries also increases, and so does the



(a) 1×1 (b) 2×2 (c) 3×3

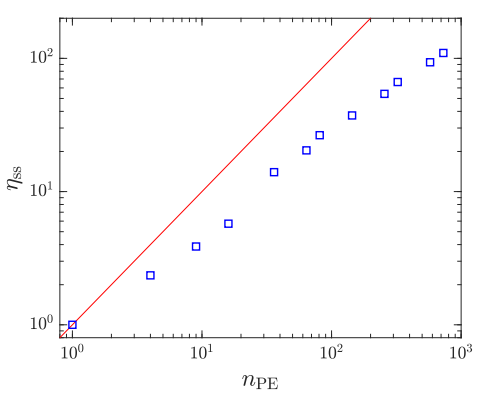


(d) System 1 (e) System 2

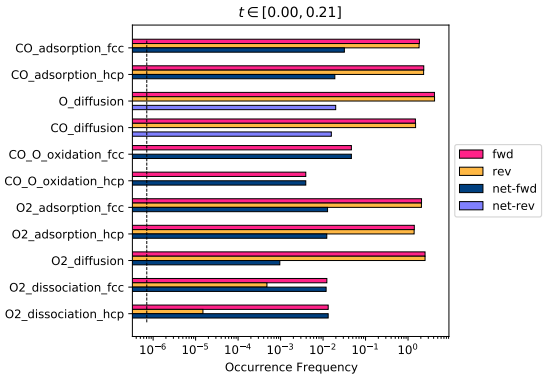
Figure 7: (a-c): In the strong-scaling analysis the number of PEs is increased while keeping the lattice size fixed, so that the workload per PE is progressively reduced. (d, e): Strong-scaling performance of the Time-Warp algorithm in *Zacros* for systems 1 and 2. Plotted is the efficiency calculated using eq. (8). Note that the left-most three points of panels (d) and (e) overlap, since they correspond to serial runs for which the efficiency is one by definition.

ratio between the number of sites that belong in halos over the total number of sites. In turn, the probability that an event will involve at least one halo site becomes higher, leading to inefficiencies due to causality violations and the ensuing roll-backs. This issue is not observed for the two larger lattices under consideration and the PE configurations investigated. However, it is expected that the strong-scaling efficiency will go through a maximum in any case, as the number of PEs is increased.

The scaling behavior is expected to be strongly affected by the amount of memory available for *stateQueue*. For instance, this amount of memory is fixed to 3.5 GB per PE for the 360×360 lattice (129,600 sites), for the 2×2 and the 4×4 PE configurations (4 versus 16 PEs). However, for



(a) System 3 - performance benchmarks



(b) System 3 - representative results

Figure 8: (a): Results of the strong-scaling benchmarks for system 3, the detailed CO oxidation model of Ref. [65], on a $2{,}592 \times 2{,}592$ lattice (more than 13.4 million sites). The efficiencies (blue squares) are calculated from eq. (8) and compared with the ideal scaling (red line). The left-most point is a serial run for which the efficiency is 1 by definition. (b): Representative results, in terms of the frequency of occurrence of the simulated events. Plotted are the forward, reverse, net-forward and net-reverse frequencies per fcc site. The net-forward frequency is the difference between forward and reverse frequency; if that difference is negative, its absolute value is plotted as the net-reverse frequency. The vertical dashed line corresponds to the (normalized) frequency of a single event.

smaller numbers of PEs, the larger domains handled would result in larger memory requirements for saving each KMC state. For this reason, it is possible that our benchmarks "favor" larger number of PE configurations, for which *stateQueue* is able to store a larger number of KMC state snapshots. A higher capacity for *stateQueue* enables "state-saving" at a higher frequency, which, in turn, minimizes the time spent in performing rollback-propagation. In fact, the best case scenario would be if one was able to store a KMC state at every single KMC step, though this is clearly infeasible due to the exceedingly high number of such steps for typical simulations. On the other hand, it is important to point out that handling large memory allocations for *stateQueue*, may also entail non-negligible overheads associated with reading and writing KMC states.

Our last set of benchmarks, on the more realistic system 3 for CO oxidation on Pd(111), are shown in fig. 8(a). We observe a steady rise in efficiency with an increase in the number of PEs, though the speedup is sublinear. Thus, if the efficiency is given as a power function of the number of PEs,

$\eta_{SS} = c \cdot n_{PE}^{\alpha}$, the exponent α calculated from a least-squares fit is equal to about 0.76 if the left-most point of the graph is neglected (for $n_{PE} = 1$, which is actually a serial run), while, if all points are included in the fit $\alpha \approx 0.74$. Indicatively, for the largest simulation performed, which employed $27 \times 27 = 729$ PEs, we obtained a speedup by a factor of about 110. This behavior is attributed to the large halo width of this system, which results in frequent causality violations that have to be resolved with rollbacks.

Finally, in fig. 8(b) we show representative results on the normalized (per cell, or per fcc site) frequency of occurrence of each elementary event included in these CO oxidation simulations. These frequencies were calculated from the aggregated number of events executed in all subdomains. We see that adsorption, desorption and diffusion events appear quasi-equilibrated. A net-forward frequency of about 4.6×10^{-2} molecules $\cdot$ site$^{-1}\cdot$ s^{-1} is observed for CO oxidation on fcc sites, which is the dominant among the two CO oxidation events. This value is in agreement with the previously reported frequency for this model under the conditions chosen (see Ref. [65], fig. 8, model 2, $p = 10^{-6}$ bar, $T = 440$ K).

4. Summary and Conclusions

On-lattice KMC methodologies have attracted significant focus in the computational catalysis field and have proven instrumental in unravelling the complex dynamic behavior of heterogeneous catalysts [15, 16]. Still though, elucidating challenging phenomena, such as catalyst reconstruction and long-range pattern formation, necessitates accurate simulations at unprecendented scales. Thus, motivated, we have successfully coupled the optimistic Time-Warp algorithm with the graph-theoretical KMC framework and implemented the approach in *Zacros*, a general-purpose on-lattice KMC code, in order to enable exact distributed KMC simulations.

The main challenge in such simulations relates to the sequential nature of the KMC algorithm, whereby lattice events exhibit causal relations. Hence, distributing the computational load among different processing elements (CPU cores, MPI processes etc.) by decomposing the lattice into subdomains is non-trivial. One is faced with the choice of either keeping PEs synchronized, which makes it easy maintain causality but can be highly inefficient computationally, or perform the subdomain simulations asynchronously but devise a way to correct any causality violations. The latter may arise during the simulation if a processor receives, via a message from a neighbor,

an instruction to perform an event in the past. Amending the simulation trajectory entails rolling back to a time before the timestamp of this past event, thereby discarding a recently simulated portion of the KMC trajectory, executing the event and simulating anew from that point onward. Of course, this rollback may itself lead to additional violations of causality, since the discarded KMC trajectory may include events that were messaged to other processors. The purpose of the Time-Warp algorithm is to handle such cascades of rollbacks via local operations, i.e. without the need for an "overseer" or global controller.

Compared to the traditional sequential GT-KMC algorithm, [17, 18] the distributed approach entails several additional operations, as part of the Time-Warp algorithm embedded in our implementation: communication of boundary-events (as messages or anti-messages, the latter encoding "undo-actions"), saving of simulation states (to be restored later if/as needed), saving of messages (in case anti-messages need to be sent later), re-winding of the simulation to an earlier time (rollback), discarding of obsolete states and messages ("fossil collection"), computation of the global virtual time and collective termination of the distributed run. We have further devised a computational scheme that can be applied to sequential GT-KMC runs, enabling them to produce results identical to those of distributed runs. This "parallel-emulation" scheme is based on a causality-related constraint inherent in the Time-Warp algorithm and allowed us to validate the correctness of our implementation.

To benchmark the performance of Time-Warp GT-KMC we have investigated both the weak- and strong-scaling, of our implementation for two systems, for which inter-domain coupling arises due to different factors. The scaling benchmarks on both systems reveal that the overhead associated with Time-Warp-related procedures can be significant; however, the distributed approach scales well with lattice size, and ends up outperforming appreciably the sequential KMC for sufficiently large lattices. For demonstration purposes, we have also performed runs for the more realistic model of Ref. [65], which captures CO oxidation dynamics on Pt(111). Our ongoing work focuses on improving memory management, especially for handling the KMC state queue, which exhibits the largest memory footprint among all datastructures in our implementation. Another direction of interest pertains to the computation of the global virtual time, and the decision variable for collective termination. Both necessitate global communication, which may adversely affect the performance of distributed runs with very large number

of processors; thus, asynchronous approaches are being considered, such as those by Mattern [64].

Overall, our Time-Warp GT-KMC implementation is quite promising and will allow the catalysis and surface-science communities to study heterogeneous catalysts at spatial and temporal scales of unprecedented magnitude. Crucially, the exact nature of the algorithm enables meaningful comparisons with experiments to be performed, making it possible to test theoretical predictions and explain complex phenomena, such as long-range pattern formation due to catalyst reconstruction. Finally, our implementation is expected to facilitate the benchmarking and further development of *approximate* algorithms for distributed KMC simulation, in the context of practical catalysis and surface-science problems.

Acknowledgments

This project has received funding from the embedded CSE program of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk) (project identifiers: eCSE01-001, eCSE10-08, eCSE01-13), the Leverhulme Trust (project RPG-2017-361) and from the European Union's Horizon 2020 research and innovation programme under grant agreement No 814416.

The authors acknowledge the use of the UCL Research Software Development Service (RSD@UCL), as well as the UCL High Performance Computing (HPC) Facilities (Myriad@UCL, Grace@UCL, and Kathleen@UCL) and associated support services in the completion of this work. The provision of computational resources by the UK Materials and Molecular Modelling Hub (specifically access to HPC facility Thomas), which is partially funded by EPSRC (EP/P020194/1 and EP/T022213/1), is also gratefully acknowledged.

References

[1] D. T. Gillespie, A general method for numerically simulating the stochastic time evolution of coupled chemical reactions, Journal of Computational Physics 22 (4) (1976) 403–434.

[2] D. T. Gillespie, Exact stochastic simulation of coupled chemical reactions, The Journal of Physical Chemistry 81 (25) (1977) 2340–2361.

[3] A. Slepoy, A. P. Thompson, S. J. Plimpton, A constant-time kinetic Monte Carlo algorithm for simulation of large biochemical reaction networks, The Journal of Chemical Physics 128 (20) (2008) 05B618.

[4] C. C. Battaile, The kinetic Monte Carlo method: Foundation, implementation, and application, Computer Methods in Applied Mechanics and Engineering 197 (41) (2008) 3386 – 3398.

[5] M. Stamatakis, D. G. Vlachos, Unraveling the complexity of catalytic reactions via kinetic Monte Carlo simulation: Current status and frontiers, ACS Catalysis 2 (12) (2012) 2648–2663.

[6] E. Ustinov, D. Do, Application of kinetic Monte Carlo method to equilibrium systems: Vapour–liquid equilibria, Journal of Colloid and Interface Science 366 (1) (2012) 216 – 223.

[7] A. A. Franco, A. Rucci, D. Brandell, C. Frayret, M. Gaberscek, P. Jankowski, P. Johansson, Boosting rechargeable batteries R&D by multiscale modeling: Myth or reality?, Chemical Reviews 119 (7) (2019) 4569–4627.

[8] M. Apostolopoulou, R. Dusterhoft, R. Day, M. Stamatakis, M.-O. Coppens, A. Striolo, Estimating permeability in shales and other heterogeneous porous media: Deterministic vs. stochastic investigations, International Journal of Coal Geology 205 (2019) 140–154.

[9] M. Apostolopoulou, M. S. Santos, M. Hamza, T. Bui, I. G. Economou, M. Stamatakis, A. Striolo, Quantifying pore width effects on diffusivity via a novel 3D stochastic approach with input from atomistic molecular dynamics simulations, Journal of Chemical Theory and Computation 15 (12) (2019) 6907–6922.

[10] K. Reuter, First-Principles Kinetic Monte Carlo Simulations for Heterogeneous Catalysis: Concepts, Status, and Frontiers, John Wiley & Sons, Ltd, 2011, Ch. 3, pp. 71–111.

[11] M. Stamatakis, Kinetic modelling of heterogeneous catalytic systems, Journal of Physics: Condensed Matter 27 (1) (2014) 013001.

[12] W. Thiel, Computational catalysis - past, present, and future, Angewandte Chemie International Edition 53 (33) (2014) 8605–8613.

[13] M. T. Darby, S. Piccinin, M. Stamatakis, First principles-based kinetic Monte Carlo simulation in catalysis, in: Physics of Surface, Interface and Cluster Catalysis, 2053-2563, IOP Publishing, 2016, pp. 4–1 to 4–38.

[14] S. Matera, W. F. Schneider, A. Heyden, A. Savara, Progress in accurate chemical kinetic modeling, simulations, and parameter estimation for heterogeneous catalysis, ACS Catalysis 9 (8) (2019) 6624–6647.

[15] K. G. Papanikolaou, M. Stamatakis, Toward the accurate modeling of the kinetics of surface reactions using the kinetic Monte Carlo method, in: Frontiers of Nanoscience, Vol. 17, Elsevier, 2020, pp. 95–125.

[16] B. W. Chen, L. Xu, M. Mavrikakis, Computational methods in heterogeneous catalysis, Chemical Reviews (2020).

[17] M. Stamatakis, D. G. Vlachos, A graph-theoretical kinetic Monte Carlo framework for on-lattice chemical kinetics, The Journal of Chemical Physics 134 (21) (2011) 214115.

[18] J. Nielsen, M. d'Avezac, J. Hetherington, M. Stamatakis, Parallel kinetic Monte Carlo simulation framework incorporating accurate models of adsorbate lateral interactions, The Journal of Chemical Physics 139 (22) (2013) 224706.

[19] R. J. Gelten, A. P. J. Jansen, R. A. van Santen, J. J. Lukkien, J. P. L. Segers, P. A. J. Hilbers, Monte Carlo simulations of a surface reaction model showing spatio-temporal pattern formations and oscillations, The Journal of Chemical Physics 108 (14) (1998) 5921–5934.

[20] M. J. Hoffmann, S. Matera, K. Reuter, kmos: A lattice kinetic Monte Carlo framework, Computer Physics Communications 185 (7) (2014) 2138–2150.

[21] M. Leetmaa, N. V. Skorodumova, KMCLib: A general framework for lattice kinetic Monte Carlo (KMC) simulations, Computer Physics Communications 185 (9) (2014) 2340–2349.

[22] R. Réocreux, P. L. Kress, R. T. Hannagan, V. Çınar, M. Stamatakis, E. C. H. Sykes, Controlling hydrocarbon (de)hydrogenation pathways with bifunctional PtCu single-atom alloys, The Journal of Physical Chemistry Letters 11 (20) (2020) 8751–8757.

[23] A. Chutia, A. Thetford, M. Stamatakis, C. R. A. Catlow, A DFT and KMC based study on the mechanism of the water gas shift reaction on the Pd(100) surface, Physical Chemistry Chemical Physics 22 (6) (2020) 3620–3632.

[24] M. Núñez, T. Robie, D. Vlachos, Acceleration and sensitivity analysis of lattice kinetic Monte Carlo simulations using parallel processing and rate constant rescaling, The Journal of Chemical Physics 147 (16) (2017) 164103.

[25] A. Chatterjee, A. F. Voter, Accurate acceleration of kinetic Monte Carlo simulations through the modification of rate constants, The Journal of Chemical Physics 132 (19) (2010) 194101.

[26] T. Danielson, J. E. Sutton, C. Hin, A. Savara, SQERTSS: Dynamic rank based throttling of transition probabilities in kinetic Monte Carlo simulations, Computer Physics Communications 219 (2017) 149–163.

[27] E. C. Dybeck, C. P. Plaisance, M. Neurock, Generalized temporal acceleration scheme for kinetic Monte Carlo simulations of surface catalytic processes by scaling the rates of fast reactions, Journal of Chemical Theory and Computation 13 (4) (2017) 1525–1538.

[28] M. Andersen, C. P. Plaisance, K. Reuter, Assessment of mean-field microkinetic models for CO methanation on stepped metal surfaces using accelerated kinetic Monte Carlo, The Journal of Chemical Physics 147 (15) (2017) 152705.

[29] A. Pedersen, J.-C. Berthet, H. Jónsson, Simulated annealing with coarse graining and distributed computing, in: International Workshop on Applied Parallel Computing, Springer, 2010, pp. 34–44.

[30] S. T. Chill, M. Welborn, R. Terrell, L. Zhang, J.-C. Berthet, A. Pedersen, H. Jonsson, G. Henkelman, EON: Software for long time simulations of atomic scale systems, Modelling and Simulation in Materials Science and Engineering 22 (5) (2014) 055002.

[31] M. Andersen, C. Panosetti, K. Reuter, A practical guide to surface kinetic Monte Carlo simulations, Frontiers in Chemistry 7 (2019) 202.

[32] X.-M. Cao, Z.-J. Shao, P. Hu, A fast species redistribution approach to accelerate the kinetic Monte Carlo simulation for heterogeneous catalysis, Physical Chemistry Chemical Physics 22 (2020) 7348–7364.

[33] A. P. J. Jansen, An introduction to kinetic Monte Carlo simulations of surface reactions, Vol. 856, Springer, 2012.

[34] T. Schulze, Kinetic Monte Carlo simulations with minimal searching, Physical Review E 65 (3) (2002) 036704.

[35] A. Chatterjee, D. G. Vlachos, An overview of spatial microscopic and accelerated kinetic Monte Carlo methods, Journal of Computer-Aided Materials Design 14 (2) (2007) 253–308.

[36] F. Hess, Efficient implementation of cluster expansion models in surface kinetic Monte Carlo simulations with lateral interactions: Subtraction schemes, supersites, and the supercluster contraction, Journal of Computational Chemistry 40 (30) (2019) 2664–2676.

[37] S. Ravipati, M. d'Avezac, J. Nielsen, J. Hetherington, M. Stamatakis, A caching scheme to accelerate kinetic Monte Carlo simulations of catalytic reactions, The Journal of Physical Chemistry A (2020).

[38] G. D. Savva, M. Stamatakis, Comparison of queueing data-structures for kinetic Monte Carlo simulations of heterogeneous catalysts, The Journal of Physical Chemistry A 124 (38) (2020) 7843–7856.

[39] G. Ertl, Reactions at surfaces: From atoms to complexity (Nobel lecture), Angewandte Chemie International Edition 47 (19) (2008) 3524–3535.

[40] F. Schüth, B. Henry, L. D. Schmidt, Oscillatory reactions in heterogeneous catalysis, in: Advances in Catalysis, Vol. 39, Elsevier, 1993, pp. 51–127.

[41] B. D. Lubachevsky, Efficient parallel simulations of dynamic Ising spin systems, Journal of Computational Physics 75 (1) (1988) 103 – 122.

[42] S. G. Eick, A. G. Greenberg, B. D. Lubachevsky, A. Weiss, Synchronous relaxation for parallel simulations with applications to circuit-switched networks, ACM Trans. Model. Comput. Simul. 3 (4) (1993) 287–314.

[43] D. R. Jefferson, Virtual time, ACM Transactions on Programming Languages and Systems (TOPLAS) 7 (3) (1985) 404–425.

[44] M. Merrick, K. A. Fichthorn, Synchronous relaxation algorithm for parallel kinetic Monte Carlo simulations of thin film growth, Physical Review E 75 (1) (2007) 011606.

[45] G. Nandipati, Y. Shim, J. G. Amar, A. Karim, A. Kara, T. S. Rahman, O. Trushin, Parallel kinetic Monte Carlo simulations of Ag(111) island coarsening using a large database, Journal of Physics: Condensed Matter 21 (8) (2009) 084214.

[46] Y. Shim, J. G. Amar, Semirigorous synchronous sublattice algorithm for parallel kinetic Monte Carlo simulations of thin film growth, Physical Review B 71 (12) (2005) 125432.

[47] E. Martínez, J. Marian, M. H. Kalos, J. M. Perlado, Synchronous parallel kinetic Monte Carlo for continuum diffusion-reaction systems, Journal of Computational Physics 227 (8) (2008) 3804–3823.

[48] M. Herlihy, J. E. B. Moss, Transactional memory: Architectural support for lock-free data structures, in: Proceedings of the 20th annual international symposium on Computer architecture (ISCA '93), Vol. 21, 1993, pp. 1–12.

[49] C. Garcia Cardona, V. Tikare, S. J. Plimpton, Parallel simulation of 3D sintering, International Journal of Computational Materials Science and Surface Engineering 4 (1) (2011) 37–54.

[50] M. Shirazi, S. D. Elliott, Atomistic kinetic Monte Carlo study of atomic layer deposition derived from density functional theory, Journal of Computational Chemistry 35 (3) (2014) 244–259.

[51] E. R. Homer, V. Tikare, E. A. Holm, Hybrid potts-phase field model for coupled microstructural–compositional evolution, Computational Materials Science 69 (2013) 414–423.

[52] W. Li, M. Soshi, Modeling analysis of grain morphologies in directed energy deposition (DED) coating with different laser scanning patterns, Materials Letters 251 (2019) 8–12.

[53] M. Van den Bossche, Kinetic Monte Carlo modeling of the catalytic hydrogenation of benzene on pt(111), Master's thesis, Universiteit Gent, Ghent, Belgium (2012).

[54] T. Oppelstrup, D. R. Jefferson, V. V. Bulatov, L. A. Zepeda-Ruiz, SPOCK: Exact parallel kinetic Monte-Carlo on 1.5 million tasks, in: Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '16), 2016, pp. 127–130.

[55] C. D. Carothers, K. S. Perumalla, R. M. Fujimoto, Efficient optimistic parallel simulations using reverse computation, ACM Transactions on Modeling and Computer Simulation 9 (3) (1999) 224–253.

[56] A. Nakano, R. K. Kalia, P. Vashishta, T. J. Campbell, S. Ogata, F. Shimojo, S. Saini, Scalable atomistic simulation algorithms for materials research, Scientific Programming 10 (2002) 263–270.

[57] S. Wang, B. Temel, J. Shen, G. Jones, L. Grabow, F. Studt, T. Bligaard, F. Abild-Pedersen, C. Christensen, J. Nørskov, Universal Brønsted-Evans-Polanyi relations for C-C, C-O, C-N, N-O, N-N, and O-O dissociation reactions, Catalysis Letters 141 (3) (2011) 370–373.

[58] C. Wu, D. Schmidt, C. Wolverton, W. Schneider, Accurate coverage-dependence incorporated into first-principles kinetic models: Catalytic NO oxidation on Pt(111), Journal of Catalysis 286 (2012) 88 – 94.

[59] H. Eyring, The activated complex in chemical reactions, The Journal of Chemical Physics 3 (2) (1935) 107–115.

[60] H. Haramoto, M. Matsumoto, F. Panneton, P. L'Ecuyer, Efficient jump ahead for $\mathbb{F}_2$ linear random number generators, INFORMS Journal on Computing 20 (2008) 385–390.

[61] D. E. Knuth, The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms, Addison Wesley Longman Publishing Co., Inc., USA, 1997.

[62] R. M. Fujimoto, Parallel discrete event simulation, Communications of the ACM 33 (10) (1990) 30–53.

[63] D. Richardson, Terminating parallel discrete event simulations, Ph.D. thesis, Virginia Tech (1991).

[64] F. Mattern, Efficient algorithms for distributed snapshots and global virtual time approximation, Journal of Parallel and Distributed Computing 18 (4) (1993) 423–434.

[65] S. Piccinin, M. Stamatakis, Steady-state CO oxidation on Pd(111): First-principles kinetic Monte Carlo simulations and microkinetic analysis, Topics in Catalysis 60 (2017) 141–151.

[66] S. Piccinin, M. Stamatakis, CO oxidation on Pd(111): A first-principles based kinetic Monte Carlo study, ACS Catalysis 4 (2014) 2143–2152.

[67] M. Stamatakis, S. Piccinin, Rationalising the relation between adlayer structure and observed kinetics in catalysis, ACS Catalysis 6 (2016) 2105–2111.

Supplementary Information:

Coupling the Time-Warp algorithm with the Graph-Theoretical Kinetic Monte Carlo framework for distributed simulations of heterogeneous catalysts

1 Priority among boundary-events with identical occurrence times

In our discussion of the Time-Warp algorithm in the main manuscript it was assumed that no two events can have the same occurrence times, and the priority among events is solely dictated and decided by time; an event with lower occurrence time has higher priority. This priority rule is used in the Time-Warp algorithm in different places, such as sorting messages in the *messgQueue*, sorting states in the *stateQueue*, checking if there is a causality violation, and updating the priority-straggler in a rollback-propagation.

However, it is possible that events have equal occurrence times. This may happen due to precision errors, e.g. if the KMC time advancements (also referred to as inter-arrival or waiting times) for two (or more) events are so small that when added to the current KMC time t_{KMC}, the resulting absolute occurrence times of these events are all equal to t_{KMC}. Another factor that may result to equal occurrence times for different events has to do with the discrete nature of the random numbers (deviates) produced by any (pseudo-)random number generator (which again is related to finite precision). Thus, it is always possible that the generator returns the same deviate for two instances of a reaction step that has constant propensity. In this case, one again ends up with two different lattice events that have the same occurrence times. The probability of this issue arising scales with the lattice size and the length of the simulation. In such cases, additional criteria are needed to decide the priority among boundary-events.

Algorithm 1: Priority among boundary-events with identical occurrence times

Data: Time-stamp (double), serial-number (s.no.) (integer), and rank-owner (integer) of each of the two boundary-events (event-1 and event-2) among which the priority is being decided

Result: Priority decision among the two boundary-events

```
1  begin
2      if events have different occurrence times then
3          if time-stamp of event-1 < time-stamp of event-2 then
4              Event-1 has higher priority;
5          else
6              Event-2 has higher priority;
7      else
           // Both events, event-1 and event-2, have identical times
8          if events have different s.no. values then
9              if s.no. of event-1 < s.no. of event-2 then
10                 Event-1 has higher priority;
11             else
12                 Event-2 has higher priority;
13         else
               // Both events, event-1 and event-2, have identical times and s.no.
                  values
14             if rank of owner of event-1 < rank of owner of event-2 then
15                 Event-1 has higher priority;
16             else
17                 Event-2 has higher priority;
```

To address such cases, each executed boundary-event is assigned a serial number (s.no.), which is a non-negative integer and is also communicated as part of the corresponding message (i.e. is a field in that message, in addition to the fields of the "Message" data-structure). The intention for the s.no. is to convey additional information (for equal time-stamp boundary-events) about the order/priority in which events are supposed to be executed. For a foreign-event, s.no. is assigned by the sender, and the receiver simply reads its value from the received message. For a boundary-event handled by the PE (i.e. belonging to its *procQueue*), a simple set of rules is used to set the s.no. of the event and decide about the priority thereof against foreign-events (received messages). As discussed earlier, in every iteration of the KMC loop, a PE executes the most imminent event among those in *procQueue* and *messgQueue* (i.e. the event that results in the smallest advancement to the KMC time). If the two most imminent events out of these queues have the same timestamp, priority among them has to be decided on the basis of the s.no. Thus, an integer variable SN that captures the serial-number of the just-executed boundary-event is initialized with -1 at every new KMC time. If the imminent event from the *procQueue* is not a boundary-event, it is executed with a higher priority and SN is not advanced. However, if the imminent event from the *procQueue* is a boundary-event, a temporary variable tmpSN is assigned the value SN + 1 and compared against the s.no. of the competing foreign-event. If tmpSN is smaller than the s.no. of the foreign-event, the boundary-event is assigned higher priority and executed. In addition, the PE commits the tmpSN value to SN and the s.no. field of the message encoding that boundary-event. On the other hand, if tmpSN is larger than the s.no. of the foreign-event, the foreign-event is assigned higher priority and executed. In this case, the PE updates SN, by assigning it the s.no. of the foreign-event and moves forward.

Note that, if all executed boundary-events have distinct times, SN would be equal to 0 and it would never be used in priority decisions (the priority is decided solely on the basis of the occurrence times). Moreover, if the imminent event is a boundary-event that has a new occurrence time (different to t_{KMC}), its s.no. would be reset to 0 according to the above procedure.

Overall, this convention aims at making executed boundary-events distinguishable by their s.no. and time values and enables PEs to discern causal relations, as is necessary for the Time-Warp algorithm to operate. Still though, since serial numbers are assigned in private by each PE, it is possible that events of different PEs, e.g. a foreign-event and a boundary-event, both have equal time and s.no. values, which would lead to a conflict in priority decision. To resolve such a conflict, an additional priority rule is invoked: the higher priority event is that of the PE with the lower rank. Algorithm 1 summarizes the process of deciding the priority of execution among events unambiguously.

2 Data from scaling benchmarks

In the tables below, t_{clock} is clock time in seconds, and t_{KMC} is the final KMC time for serial runs or the GVT for distributed runs, both in units of simulated seconds. Moreover, t^* is the scaled time as defined by equation (5) of the main manuscript ($t_{\mathrm{KMC}}/t_{\mathrm{clock}}$). Finally, η, η_{ws} and η_{ss} are the efficiency factors for the serial, weak-scaling and strong-scaling runs defined by equations (6), (7) and (8) of the main manuscript.

Data from weak-scaling benchmarks for system 1 that includes CO* adsorption, desorption, and diffusion without any lateral interactions.

n_{sites}	Serial run				Distributed run				
($\times 10^4$)	t_{clock}	t_{KMC}	t^*	η	n_{PE}	t_{clock}	t_{KMC}	t^*	η_{ws}
1	3,898	12,430.3	3.1886	$1.00 \cdot 10^{0}$	1	4,490	8,203.5	1.827	0.5730
4	3,895	2,683.5	0.6889	$2.16 \cdot 10^{-1}$	4	4,486	676.8	0.151	0.0473
9	4,484	1,331.5	0.2970	$9.31 \cdot 10^{-2}$	9	4,481	427.0	0.095	0.0299
16	3,881	545.8	0.1406	$4.41 \cdot 10^{-2}$	16	4,471	298.3	0.067	0.0209
25	4,471	351.3	0.0786	$2.46 \cdot 10^{-2}$	25	4,455	192.6	0.043	0.0136
36	3,859	185.3	0.0480	$1.51 \cdot 10^{-2}$	36	4,431	176.6	0.040	0.0125
64	3,827	85.3	0.0223	$6.99 \cdot 10^{-3}$	64	4,361	134.9	0.031	0.0097
100	3,786	50.6	0.0134	$4.19 \cdot 10^{-3}$	100	4,266	146.2	0.034	0.0107
144	3,736	33.0	0.0088	$2.77 \cdot 10^{-3}$	144	4,221	148.9	0.035	0.0111
225	3,644	19.9	0.0055	$1.72 \cdot 10^{-3}$	225	4,076	139.5	0.034	0.0107
324	3,529	13.0	0.0037	$1.15 \cdot 10^{-3}$	324	3,896	128.9	0.033	0.0104
400	3,442	10.5	0.0031	$9.59 \cdot 10^{-4}$	400	3,746	126.6	0.034	0.0106
625	3,195	5.9	0.0019	$5.82 \cdot 10^{-4}$	625	3,370	112.6	0.033	0.0105
900	2,887	3.7	0.0013	$3.98 \cdot 10^{-4}$	900	2,875	91.1	0.032	0.0099

Data from weak-scaling benchmarks for system 2 that includes CO^* adsorption and desorption with first nearest neighbor interactions among CO^* adsorbates.

n_{sites}	Serial run				Distributed run				
$(\times 10^4)$	t_{clock}	t_{KMC}	t^*	η	n_{PE}	t_{clock}	t_{KMC}	t^*	η_{ws}
1	3,899	88,656.72	$2.27 \cdot 10^{1}$	$1.00 \cdot 10^{0}$	1	4,490	70,645.6	15.73	0.6920
4	3,897	14,750.69	$3.79 \cdot 10^{0}$	$1.67 \cdot 10^{-1}$	4	4,492	15,769.0	3.51	0.1544
9	4,491	2,460.85	$5.48 \cdot 10^{-1}$	$2.41 \cdot 10^{-2}$	9	4,486	11,831.2	2.64	0.1160
16	3,891	1,226.79	$3.15 \cdot 10^{-1}$	$1.39 \cdot 10^{-2}$	16	4,480	9,094.4	2.03	0.0893
25	4,483	277.85	$6.20 \cdot 10^{-2}$	$2.73 \cdot 10^{-3}$	25	4,477	7,391.2	1.65	0.0726
36	3,879	174.18	$4.49 \cdot 10^{-2}$	$1.98 \cdot 10^{-3}$	36	4,468	5,955.9	1.33	0.0586
64	3,863	57.88	$1.50 \cdot 10^{-2}$	$6.59 \cdot 10^{-4}$	64	4,417	5,148.7	1.17	0.0513
100	3,843	24.11	$6.27 \cdot 10^{-3}$	$2.76 \cdot 10^{-4}$	100	4,381	4,853.1	1.11	0.0487
144	3,817	11.87	$3.11 \cdot 10^{-3}$	$1.37 \cdot 10^{-4}$	144	4,318	4,731.3	1.10	0.0482
225	3,771	4.91	$1.30 \cdot 10^{-3}$	$5.72 \cdot 10^{-5}$	225	4,276	4,808.2	1.12	0.0495
324	3,714	1.31	$3.52 \cdot 10^{-4}$	$1.55 \cdot 10^{-5}$	324	4,195	4,251.5	1.01	0.0446
400	3,669	0.78	$2.14 \cdot 10^{-4}$	$9.39 \cdot 10^{-6}$	400	4,123	4,554.8	1.10	0.0486
625	3,541	0.23	$6.54 \cdot 10^{-5}$	$2.88 \cdot 10^{-6}$	625	3,958	4,289.4	1.08	0.0477
900	3,380	0.10	$3.01 \cdot 10^{-5}$	$1.32 \cdot 10^{-6}$	900	3,700	3,784.0	1.02	0.0450

Data from strong-scaling benchmarks for system 1 that includes CO^* adsorption, desorption and diffusion without any lateral interactions. The three blocks of data correspond to lattice sizes 360×360, 900×900, and $1{,}440 \times 1{,}440$, respectively. Runs with $n_{PE} = 1$ are performed with the serial KMC algorithm.

n_{PE}	t_{clock}	t_{KMC}	t^*	η_{ss}
1	4,460	834.002	$1.87 \cdot 10^{-1}$	1.000
4	4,473	71.524	$1.60 \cdot 10^{-2}$	0.086
9	4,471	212.492	$4.75 \cdot 10^{-2}$	0.254
16	4,476	445.995	$9.97 \cdot 10^{-2}$	0.533
25	4,470	751.063	$1.68 \cdot 10^{-1}$	0.898
36	4,465	953.763	$2.14 \cdot 10^{-1}$	1.142
64	4,470	1,816.989	$4.06 \cdot 10^{-1}$	2.174
81	4,117	2,116.483	$5.14 \cdot 10^{-1}$	2.749
100	4,434	2,364.013	$5.33 \cdot 10^{-1}$	2.851
144	4,467	2,767.635	$6.20 \cdot 10^{-1}$	3.313
225	4,410	2,450.003	$5.56 \cdot 10^{-1}$	2.971
324	4,403	3,551.588	$8.07 \cdot 10^{-1}$	4.313
400	4,365	2,175.610	$4.98 \cdot 10^{-1}$	2.666
1	4,411	74.527	$1.69 \cdot 10^{-2}$	1.000
4	4,224	2.319	$5.49 \cdot 10^{-4}$	0.032
9	3,423	6.785	$1.98 \cdot 10^{-3}$	0.117
16	4,331	18.527	$4.28 \cdot 10^{-3}$	0.253
25	4,344	32.801	$7.55 \cdot 10^{-3}$	0.447
36	4,348	40.016	$9.20 \cdot 10^{-3}$	0.545
81	4,324	152.304	$3.52 \cdot 10^{-2}$	2.085
100	4,352	212.842	$4.89 \cdot 10^{-2}$	2.895
144	4,364	438.470	$1.00 \cdot 10^{-1}$	5.946
225	4,330	712.283	$1.64 \cdot 10^{-1}$	9.736
324	4,320	1,286.298	$2.98 \cdot 10^{-1}$	17.622
400	4,330	1,606.280	$3.71 \cdot 10^{-1}$	21.955
1	4,269	24.324	$5.70 \cdot 10^{-3}$	1.000
4	2,583	0.221	$8.60 \cdot 10^{-5}$	0.015
9	2,649	0.989	$3.73 \cdot 10^{-4}$	0.066
16	3,577	2.049	$5.73 \cdot 10^{-4}$	0.101
25	3,854	6.147	$1.60 \cdot 10^{-3}$	0.280
36	3,932	8.440	$2.15 \cdot 10^{-3}$	0.377
64	4,064	19.270	$4.74 \cdot 10^{-3}$	0.832
81	3,988	27.045	$6.78 \cdot 10^{-3}$	1.190
100	4,119	40.161	$9.75 \cdot 10^{-3}$	1.711
144	4,061	74.354	$1.83 \cdot 10^{-2}$	3.214
225	4,156	159.379	$3.84 \cdot 10^{-2}$	6.732
324	4,105	316.426	$7.71 \cdot 10^{-2}$	13.528
400	4,110	464.378	$1.13 \cdot 10^{-1}$	19.830

Data from strong-scaling benchmarks for system 2 that includes CO^* adsorption and desorption with first nearest neighbor interactions among CO^* adsorbates. The three blocks of data correspond to lattice sizes 360×360, 900×900, and $1{,}440 \times 1{,}440$, respectively. Runs with $n_{PE} = 1$ are performed with the serial KMC algorithm.

n_{PE}	t_{clock}	t_{KMC}	t^*	η_{ss}
1	4,486	1,086.374	$2.42 \cdot 10^{-1}$	1.000
4	4,485	2,009.258	$4.48 \cdot 10^{-1}$	1.850
9	4,486	6,220.179	$1.39 \cdot 10^{0}$	5.725
16	4,486	13,002.530	$2.90 \cdot 10^{0}$	11.969
25	4,485	20,451.992	$4.56 \cdot 10^{0}$	18.828
36	4,452	26,301.178	$5.91 \cdot 10^{0}$	24.392
64	4,485	49,115.564	$1.10 \cdot 10^{1}$	45.216
81	4,306	54,141.791	$1.26 \cdot 10^{1}$	51.915
100	4,413	62,302.361	$1.41 \cdot 10^{1}$	58.289
144	4,467	40,044.195	$8.96 \cdot 10^{0}$	37.014
225	4,438	33,883.204	$7.63 \cdot 10^{0}$	31.522
324	4,424	33,948.556	$7.67 \cdot 10^{0}$	31.686
400	4,413	28,955.754	$6.56 \cdot 10^{0}$	27.094
1	4,453	29.146	$6.55 \cdot 10^{-3}$	1.000
4	4,365	30.204	$6.92 \cdot 10^{-3}$	1.057
9	4,386	207.718	$4.74 \cdot 10^{-2}$	7.235
16	4,303	505.521	$1.17 \cdot 10^{-1}$	17.949
25	4,426	954.577	$2.16 \cdot 10^{-1}$	32.950
36	4,425	1,361.765	$3.08 \cdot 10^{-1}$	47.014
81	4,373	5,180.792	$1.18 \cdot 10^{0}$	180.972
100	4,396	7,350.978	$1.67 \cdot 10^{0}$	255.482
144	4,431	12,898.272	$2.91 \cdot 10^{0}$	444.658
225	4,407	23,202.373	$5.26 \cdot 10^{0}$	804.257
324	4,371	32,384.591	$7.41 \cdot 10^{0}$	1,131.802
400	4,369	36,249.845	$8.30 \cdot 10^{0}$	1,267.559
1	4,385	4.542	$1.04 \cdot 10^{-3}$	1.000
4	3,605	4.507	$1.25 \cdot 10^{-3}$	1.207
9	3,952	30.687	$7.77 \cdot 10^{-3}$	7.498
16	4,099	85.724	$2.09 \cdot 10^{-2}$	20.191
25	4,189	179.106	$4.28 \cdot 10^{-2}$	41.283
36	4,187	261.495	$6.25 \cdot 10^{-2}$	60.299
64	4,286	515.776	$1.20 \cdot 10^{-1}$	116.190
81	4,257	907.656	$2.13 \cdot 10^{-1}$	205.879
100	4,316	1,239.003	$2.87 \cdot 10^{-1}$	277.150
144	4,264	2,424.878	$5.69 \cdot 10^{-1}$	549.013
225	4,306	5,586.424	$1.30 \cdot 10^{0}$	1,252.654
324	4,296	8,634.549	$2.01 \cdot 10^{0}$	1,940.365
400	4,303	11,278.285	$2.62 \cdot 10^{0}$	2,530.832

Data from strong-scaling benchmarks for system 3 that includes 22 elementary events capturing CO oxidation dynamics on Pd(111). t_{clock} is in seconds. t_{KMC} is GVT for distributed runs. The block of data corresponds to the 2,592 $\times$ 2,592 lattice.

n_{PE}	t_{clock}	t_{KMC}	t^*	η_{ss}
1	25,424.7	$1.21 \cdot 10^{-3}$	$4.75 \cdot 10^{-8}$	1.000
4	10,570.9	$1.18 \cdot 10^{-3}$	$1.12 \cdot 10^{-7}$	2.350
9	23,175.3	$4.26 \cdot 10^{-3}$	$1.84 \cdot 10^{-7}$	3.869
16	18,506.6	$5.04 \cdot 10^{-3}$	$2.72 \cdot 10^{-7}$	5.741
36	35,418.6	$2.35 \cdot 10^{-2}$	$6.64 \cdot 10^{-7}$	13.988
64	38,037.0	$3.68 \cdot 10^{-2}$	$9.68 \cdot 10^{-7}$	20.390
81	38,768.6	$4.87 \cdot 10^{-2}$	$1.26 \cdot 10^{-6}$	26.498
144	39,902.4	$7.06 \cdot 10^{-2}$	$1.77 \cdot 10^{-6}$	37.282
256	41,078.1	$1.06 \cdot 10^{-1}$	$2.57 \cdot 10^{-6}$	54.142
324	41,137.5	$1.30 \cdot 10^{-1}$	$3.15 \cdot 10^{-6}$	66.434
576	41,249.6	$1.83 \cdot 10^{-1}$	$4.44 \cdot 10^{-6}$	93.478
729	41,165.8	$2.14 \cdot 10^{-1}$	$5.21 \cdot 10^{-6}$	109.785