THE JOURNAL OF PHYSICAL CHEMISTRY A

A: New Tools and Methods in Experiment and Theory

Comparison of Queueing Data-Structures for Kinetic Monte Carlo Simulations of Heterogeneous Catalysts

Giannis D. Savva, and Michail Stamatakis

J. Phys. Chem. A, **Just Accepted Manuscript** • DOI: 10.1021/acs.jpca.0c06871 • Publication Date (Web): 01 Sep 2020

Downloaded from pubs.acs.org on September 4, 2020

Comparison of Queueing Data-Structures for Kinetic Monte Carlo Simulations of Heterogeneous Catalysts

Giannis D. Savva and Michail Stamatakis[*]

Thomas Young Centre and Department of Chemical Engineering, University College London, Roberts Building, Torrington Place, London WC1E 7JE, UK

[*] email: m.stamatakis@ucl.ac.uk

Abstract

On-lattice Kinetic Monte Carlo (KMC) is a computational method used to simulate (among others) physico-chemical processes on catalytic surfaces. The KMC algorithm propagates the system through discrete configurations by selecting (with the use of random numbers) the next elementary process to be simulated, e.g. adsorption, desorption, diffusion or reaction. An implementation of such a selection procedure is the *first-reaction method* in which all realizable elementary processes are identified and assigned a random occurrence time based on their rate constant. The next event to be executed will then be the one with the minimum inter-arrival time. Thus, a fast and efficient algorithm for selecting the most imminent process and performing all the necessary updates on the list of realizable processes post-execution, is of great importance. In the current work, we implement five data-structures to handle the elementary process queue during a KMC run: an unsorted list, a binary heap, a pairing heap, a 1-way skip list, and finally, a novel 2-way skip list with a mapping array specialized for KMC simulations. We also investigate the effect of compiler optimizations on the performance of these data-structures on three benchmark models, capturing CO-oxidation, a simplified water-gas shift mechanism, and a temperature programmed desorption run. Excluding the least efficient and impractical for large problems unsorted list, we observe a 3× speedup of the binary or pairing heaps (most efficient) compared to the 1-way skip list (least efficient). Compiler optimizations deliver a speedup of up to 1.8×. These benchmarks provide valuable insight on the importance of, often-overlooked, implementation-related aspects of KMC simulations, such as the queueing data-structures. Our results could be particularly useful in guiding the choice of data-structures and algorithms that would minimize the computational cost of large-scale simulations.

Keywords: queueing systems, binary heap, pairing heap, skip list, first reaction method

1. Introduction

A chemical reaction can be seen as the conversion of interacting molecules from reactant to product species via rearrangement of their atoms. The simulation of the dynamic evolution of such reactive processes can in principle be achieved by atomistic simulation methods like molecular dynamics (MD), whereby the numerical integration of Newton's equations of motion resolves the trajectory of each atom in the system, under the effect of the chosen interatomic potential and boundary conditions. However useful this method has been so far,[1] it is limited by the fact that accurate and stable integration requires extremely small time steps (~10^{-15} s) in order to capture atomic vibrations which, in turn, limits the accessible timescales to a few microseconds.[2] Therefore, phenomena or processes that take place on longer timescales, e.g. slow (infrequent) chemical reactions, are inaccessible by MD methods.

Kinetic Monte Carlo (KMC) is a stochastic computational method that, unlike MD, does not resolve the entire trajectory followed by individual atoms and molecules.[3,4] In KMC, the motion of the molecules is spatially coarse-grained in the sense that vibrations, which are not of interest, are not resolved at all. What is of interest, especially in the field of catalysis, is the occupancy of each site on the catalytic surface, represented by a suitably defined lattice, and the statistics of transitions from reactant to product states. This crossing of states is simulated explicitly in MD along with all the vibrations that lead to it. In KMC instead, only the initial and final states are considered and information about the transition is provided as an input in the form of a rate constant. Given an initial surface (lattice) configuration, with occupied and vacant sites, the chemical system is propagated in time through a discrete set of configurations with known transition rates among the various states. This different simulation scope enables KMC to reach much longer timescales than what is achievable by MD methods.[5,6] The stochastic nature of this method lies in the fact that only the transition rates among the different configurations are known and not their exact occurrence time; the latter is in fact a random variable that depends on the transition rate constant.

In general, kinetic Monte Carlo algorithms undertake the simulation of a trajectory that stems from the properties of the system under study. Regardless of implementation specifics, at every step of a KMC simulation we have a list of all the possible events that may happen on the lattice. We, then, need to randomly select an event along with its time of occurrence and execute it. Finally, we perform the necessary updates on both the lattice and the list of possible events. The bookkeeping of these procedures, namely the selection of a process, its execution and the post-execution updates, depends on the algorithmic implementation and is usually carried out using appropriate data-structures. For instance, Jansen has used a binary search tree to store the reactions while simulating systems with time-dependent rate constants.[7] Later, Lukkien and co-workers developed three KMC methods and have used a binary tree to store the possible reactions.[8] Along these lines, Gibson and Bruck developed another KMC algorithm relying on an indexed priority queue,[9] which enables the retrieval of an arbitrary reaction in constant time. The same authors have also proposed the use of a complete tree data-structure with a divide-and-conquer search algorithm in order to achieve an execution time that scales logarithmically with respect to the total available reactions on the lattice. More recently, Chatterjee and Vlachos[10] reviewed the available KMC algorithms along with the efficiency of the various implementations. They report that implementations involving linear search on the data-structure scale as $O(N_{queue})$ with N_{queue} being the total number of elements in the queueing data-structure. They first report further extensions to the linear search such as the two-level and n-level linear search that scale as $O(\sqrt{N_{queue}})$ and $nO(\sqrt[n]{N_{queue}})$ respectively. The binary and hash-table search are discussed as well. Chatterjee and Vlachos[10] conclude that sophisticated search algorithms can result in significant computational savings. In addition, due to the fact that reaction occurrence in a lattice KMC affects only a limited number of neighboring sites, updates to the corresponding queueing data-structures are limited ("local updates"). Thus, exploiting this property also leads to computationally efficient simulations.

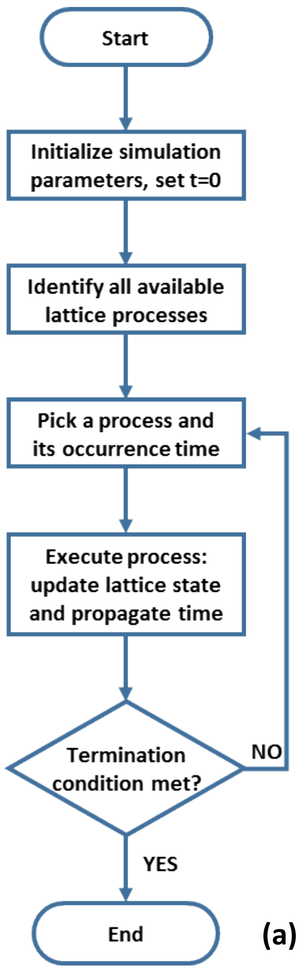
In the studies just noted, the main motivation behind the development of new data-structures for KMC was to improve the efficiency of the simulation, and this was shown to be achievable using some variant of a divide-and-conquer strategy (binary trees are essentially implementations based on this concept). However, a detailed comparative study on the performance of a wider variety and more modern data-structures as queueing systems for KMC is lacking. Such comparisons could be of interest to developers of kinetic Monte Carlo software, such as *Zacros*,[11,12] SPPARKS,[13,14] KMCLib,[15] kmos,[16] EON,[17] CARLOS,[18] MonteCoffee,[19] MoCKa[20] for materials and catalysis simulations, or Dizzy,[21] StochPy,[22] CERENA[23] for biological reaction simulations, but also for users, who would like to adopt the most efficient implementations for their simulations. In this work, we present the implementation and comparison of four data-structures (i.e. the unsorted list, the binary heap, the pairing heap and the skip list), as alternative queueing systems for KMC simulations, and we further develop the skip list to address specific needs of such simulations. These four+one queueing systems are incorporated in the Graph-Theoretical kinetic Monte Carlo (GT-KMC) framework as implemented by the KMC software package *Zacros*.[12,24] The performance of the queueing systems is evaluated using three different chemical reaction models: the Ziff-Gulari-Barshad (ZGB) CO oxidation model,[25] a simplified water-gas shift (WGS) model based on Ref. 26 and a temperature-programmed-desorption (TPD) model of CO from a pure Cu surface.[27] As a comparison measure, we use the time taken to simulate a fixed number of KMC steps with each of the different approaches. Aiming to assess the core performance of our algorithms in a variety of settings, we compile our code with and without optimizations and we present and compare results for both cases.

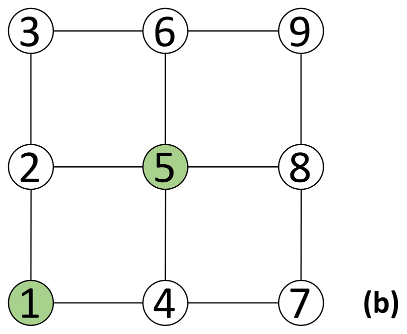
In the following sections, we present briefly the KMC method implemented by *Zacros* and the development and implementation methodology for the data-structures (Section 2). We then proceed with the description of the chemical models used to benchmark our queueing systems along with our results in Section 3. Finally, Section 4 summarizes our results as well as their importance, while we provide some useful generic guidelines on the choice of data-structures depending on the chemical system under study.

2. Methodology

2.1. Kinetic Monte Carlo

The backbone of the on-lattice KMC method, schematically shown in Figure 1(a), consists of repeatedly identifying and randomly selecting a realizable elementary event among adsorption, desorption, diffusional hops and reaction on the surface as listed in Figure 1(c). Two widely used and statistically equivalent KMC implementations are the Direct Method and First Reaction Method, introduced by Gillespie in his seminal paper,[28] on the stochastic simulation of coupled chemical reactions. The idea behind the Direct Method is to use two random numbers, one that picks the reaction r that is going to happen next and another that determines the time, τ^*, when this reaction is going to occur. On the other hand, the First Reaction Method makes use of the intuitive idea that the next reaction to be executed has to be the most imminent one, namely the one with the smallest waiting time. Therefore, a tentative reaction time τ_i is generated for every possible reaction i. Then, the reaction to execute and its occurrence time are both determined by picking the minimum time, τ^*, among all the reaction times τ_i. These two methods are able to stochastically simulate a system exactly and are equivalent,[28] in the sense that the pair (r, τ^*) in both methods is drawn from the same probability distribution. *Zacros*[12,24] implements the *First Reaction Method*, which is described in more detail below.


Start
Initialize simulation parameters, set t=0
Identify all available lattice processes
Pick a process and its occurrence time
Execute process: update lattice state and propagate time
Termination condition met?
NO
YES
End
(a)



Label	Lattice Process
1	Desorption from site 1
2	Diffusion 1 $\rightarrow$ 2
3	Diffusion 1 $\rightarrow$ 4
4	Adsorption to site 2
5	Adsorption to site 3
6	Adsorption to site 4
7	Desorption from site 5
8	Diffusion 5 $\rightarrow$ 6
9	Diffusion 5 $\rightarrow$ 8
10	Diffusion 5 $\rightarrow$ 4
11	Diffusion 5 $\rightarrow$ 2
12	Adsorption to site 6
13	Adsorption to site 7
14	Adsorption to site 8
15	Adsorption to site 9

(c)

Figure 1: (a) Flowchart of the KMC method. (b) Non-periodic square lattice with sites 1 and 5 being occupied by an adsorbate. (c) List of all realizable elementary events of the lattice shown in (b).

A KMC simulation is initiated by providing *Zacros* with an appropriate representation of the catalytic surface in the form of a lattice (e.g. as the one shown in Figure 1(b)), as well as an ensemble of chemical reactions along with the relevant reaction rates and energetic contributions to the lattice. Next, all elementary events are identified, as listed in Figure 1(c), and for each one of them a random occurrence time t_i is generated as

$$t_i = t_{cur} - \frac{1}{k_i}\ln(u) \tag{1}$$

where t_{cur} is the current KMC time (in the beginning t_{cur} = 0), k_i is the kinetic rate constant of that particular elementary event and *u* is a uniform random number in the range (0, 1). Subsequently, the minimum time among all the reaction times is retrieved and the reaction that it corresponds to is executed. Reaction execution includes updating the state of the lattice and the occurrence times of the affected elementary events only.[9,10]

Computationally, one has to create a process queue to store the absolute occurrence times, generated via equation (1), of all the available processes, and identify their minimum along with the individual event it corresponds to. Then, the chosen event, *j*, is executed and the lattice state and the process queue entries are updated. Updating the process queue includes: (i) removing the elementary processes that are no longer realizable, (ii) detecting the new possible elementary processes, assigning each of them a random time and inserting it in the queue, and (iii) modifying the inter-arrival time of processes for which the rate constant changed due to adsorbate-adsorbate lateral interactions as a result of the execution of process *j*. This procedure is iteratively repeated until a termination condition is met, e.g. a final KMC time is reached or a predefined number of KMC steps have been simulated. If numerous lattice processes are stored in the queue and long-range interactions are included in the modelling mechanism, inserting, deleting and updating elements may become computationally expensive.

Throughout the course of a KMC simulation, all the elementary lattice processes have a unique identifier, their label, which is generated upon their insertion in the process queue. Moreover, each lattice process holds all the information needed for its execution: (i) the reactants involved and (ii) the products and the lattice sites on which the former have to be added post-execution. These lattice processes are always referenced using their unique label and for that reason the update and removal operations of the process queue, as highlighted in the previous paragraph, take as input the label of the lattice process that needs to be updated or removed. Such labels constitute extra information that needs to accompany the occurrence time of each lattice process; whenever not possible to infer the label indirectly, it has to be stored in the queueing system in a way described in more detail in the following section.

2.2. Queueing system data-structures

In this subsection, we present the main architecture of the data-structures implemented as queueing systems. For the skip list data-structure, we elaborate on the motivation for further development as well as the modifications to the original data-structure. In the following discussion, we refer to the *occurrence times* as *elements*, unless a different definition is given.

2.2.1. Unsorted list

The simplest way to keep track of all the occurrence times is to store them in a one-dimensional array as they are being generated via equation (1). The array index serves as the elementary event identifier (label) since it uniquely describes an event. The retrieval of the minimum element requires to iterate over all the entries in the array, therefore, this operation scales as $O(N)$

where N is the number of elements the array contains. Given its label, removing an element is a constant time operation, namely, the time required to perform it does not depend on the size of the array. In addition, if no “gaps” are allowed in the array, that is, upon a removal, we move the last element to the position of the just-removed-element, then the insertion of a new element is done in constant time, since it is always inserted at the back of the list. Similarly, the modification of an existing value in the list is a constant time operation.

2.2.2. Binary heap

The binary heap (chapter 6 in Ref. 29) is a data-structure that belongs to the broader family of binary tree structures, in which each node has at most two children. In addition, the binary heap is always complete in the sense that all levels, with the exception of the bottom one, are fully filled with elements (Figure 2). Moreover, every node in the binary heap has “priority” over all its children, a property that makes the binary heap partially ordered. However, no other conclusions may be inferred regarding elements in different branches and different levels. Due to the partial order, the minimum element is always on the top node of the binary heap and its retrieval is a constant time operation.

A new element is inserted as a new leaf at the bottom level to the left-most position. If the new element has priority over its parent, the two elements are swapped (refer to the supporting information, section I, for graphical representation of the operation). These swaps restore the partial order of the binary tree in $O(\log_2 N)$ computational time. On removal of an element, the rightmost node of the bottom level replaces the removed node. Since it is likely that partial order is now violated, the previously last element is swapped by its current parent or the minimum of its children depending on the priority it has over them. Likewise, on update, the new value floats upwards or sinks down based on its priority over its parent and children nodes. Both operations exhibit $O(\log_2 N)$ execution time scaling.

Since the binary tree is complete, it can be implemented using an array instead of pointers (see supporting information, section I); therefore, each element (occurrence time) has a unique index that serves as the element identifier, just like in the case described in subsection 2.2.1. Nevertheless, this array index does not correspond to the lattice process label. Other supporting arrays are used to ensure a one-to-one connection between a lattice process, its occurrence time and its position on the binary heap, equivalently, its position on the 1-D array representing the binary heap (refer to supporting information, section I for more details). Given its label, any element is retrievable in constant time and therefore no additional overhead is incurred on the update and removal operations.

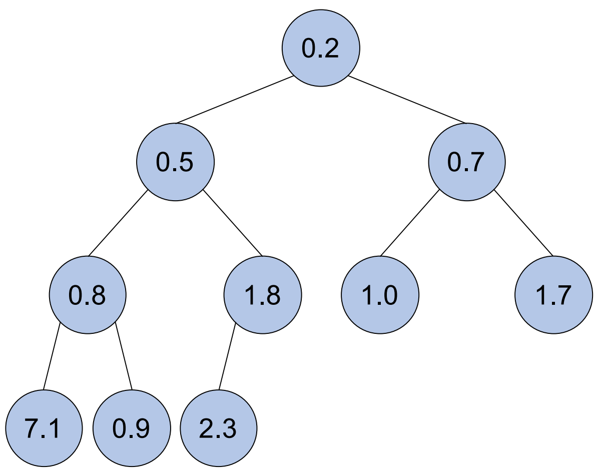


Figure 2: A binary heap with 10 elements where the bottom level is incomplete.

2.2.3. Pairing heap

The pairing heap, introduced by Fredman and co-workers,[30] is a heap data-structure in which elements are stored partially sorted. Unlike the binary heap, the pairing heap allows for an arbitrary number of children per node as illustrated in Figure 3. However, due to this shape flexibility, the notion of *completeness* does not apply to the pairing heap; therefore, from the implementation point of view, a pairing heap requires pointers to connect the nodes. In addition, for an efficient implementation, the *binary tree representation* (chapter 10.4 in Ref. 29, 30) is often used (see supporting information, section II, for details). Direct access to elements is provided by an indexing array with pointers, in a similar manner as the indexed priority queue of Gibson and Bruck.[9]

To insert a new element in a pairing heap, we compare its value with the value of the top node. If the value of the new element is smaller than the top node, then the new element becomes the top node, otherwise the new element is added as the left-most child of the top node. The insertion operation involves just a numerical comparison, so it is a constant time operation. Due to the partial order property, finding the minimum is a constant time operation as well.

The deletion operation requires reconnecting together all the children (single nodes and entire subtrees) of the deleted node. However, the order in which the tree recombination is carried out is crucial for performance as discussed in Ref. 30. Five different variants are presented in Ref. 30 and in our implementation we have chosen the two-pass-in-opposite-directions variant (as illustrated by Fig. 7 in Ref. 30). It can be proved[30] that the removal operation runs in $O(\log_2 N)$ amortized time.

The update operation, namely, the modification (increase or decrease) of an existing value of the pairing heap, can be implemented as a removal following an insertion operation. Therefore, it also has amortized time complexity $O(\log_2 N)$. Observing that increasing a value may not violate the partial order property if the node of interest has no children, we are able to perform the update operation in constant time at the cost of checking whether the node whose value is to be decreased has a child or not.

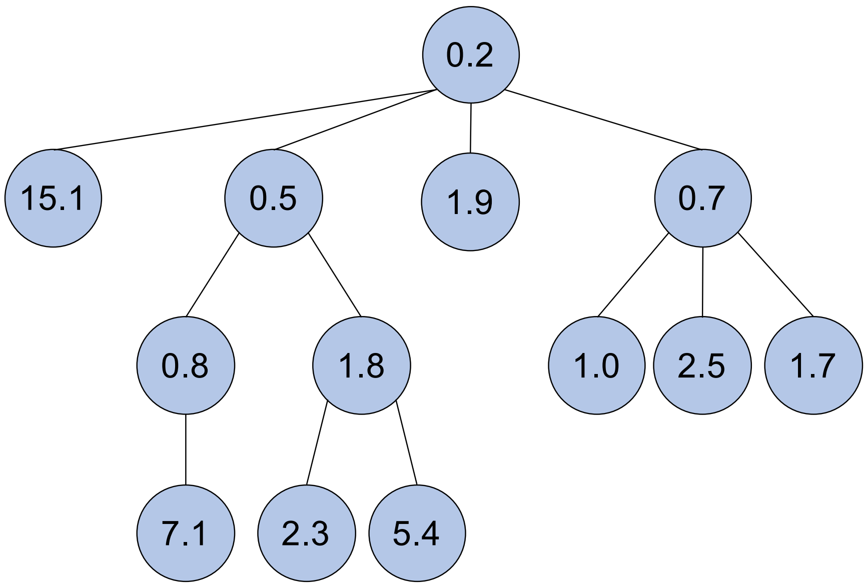


Figure 3: A pairing heap representation.

2.2.4. Skip list

The skip list is a fully ordered, linked list based data-structure proposed by W. Pugh as a probabilistic and simpler alternative to balanced trees.[31] Figure 4 illustrates a simple skip list containing six elements. The bottom layer is an ordinary linked list, where every node points to its next one in the queue. Removing an element from a linked list requires the traversal of all the nodes preceding the element being sought for deletion. Similarly, inserting a new element requires the identification of its proper location. Therefore, the main idea behind the development of this data-structure was to find a way to traverse a linked list as quickly as possible using a divide-and-conquer approach instead of sequential access.

This functionality is achieved by allowing the nodes to have variable height thus forming interconnected linked lists, which compose the levels of the skip list. The "higher-level" linked lists are sparser, as they skip progressively more and more elements of the original list. Thus, these different levels provide a shorter path (i.e. faster access) to the nodes further down the list, thereby accelerating the search, remove and insert operations. These operations are performed hierarchically, starting from the top linked list and continuing to the lower levels, enabling the skipping of a significant number of intermediate nodes. The above strategy (essentially a divide-and-conquer implementation) improves the amortized execution time of the search, insertion and deletion operations to $O(\log_2 N)$, where N is the number of elements in the skip list. The update operation, seen as a combination of a removal plus an insertion, also has an amortized execution time of $O(\log_2 N)$.

Each node stores the key value and one pointer per level, pointing to the next node in the linked list of that level. For our purposes, the key value is the time of occurrence of an elementary lattice process. As already mentioned, the skip list's nodes may have heights greater than one, whose unitary increments are determined by independent identically distributed Bernoulli random variables with success probability $p \in (0,1)$. Therefore, the resulting heights follow the geometric distribution with parameter *p*. Following the definition, the expected maximum number of levels an individual node may have is given as:

$$H_N = \log_{1/p} N \tag{2}$$

Most often, a value of p=1/2 is used, which offers a good balance between speedup and memory requirements. Different applications may benefit from choosing a different value for *p*, for example 1/4 or 1/e, as discussed in Ref. 31, by trading the search cost with storage cost.

The search operation starts from the topmost level of the head element and follows the forward pointers as long as the element being searched for is not overshot, supposing that it exists in the skip list. The search continues to the immediate lower level until the bottom level is reached and the element is found. If the element does not exist in the list, the search operation finds its predecessor and therefore the appropriate position to insert it. Deletion also uses the search operation to find the element to be deleted.

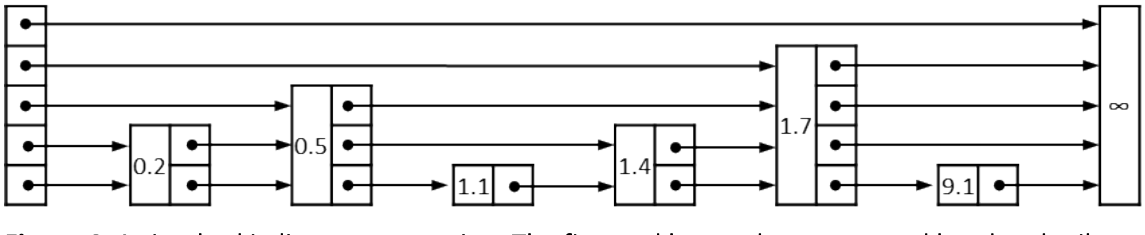


Figure 4: A simple skip list representation. The first and last nodes are termed head and tail respectively.

Both insertion and deletion operations require that the rightmost pointer of every level in the search path is temporarily stored in an array, which we refer to as *update_array* in subsequent sections. These pointers are used to correctly connect the new node with the existing ones upon insertion or the remaining nodes upon removal. A more thorough description of the skip list data-structure, its operations and pseudocode can be found in the original study.[31]

2.2.5. Developing skip list for KMC simulations

From the event scheduling perspective along the course of a KMC simulation, an efficient data-structure serving as the queueing system should incorporate effective implementations of the following operations:

a) Retrieval of the element with minimum key value. This value is the occurrence time of the next event, which is accompanied by a unique identifier / label that points to a lattice process.
b) Insertion of new elements. These new elements represent lattice processes that are created as a result of the previously occurred event.
c) Deletion of existing elements identified by their label (instead of their occurrence time as in the original skip list data-structure). The elements to be deleted represent the lattice processes that are no longer valid/realizable, for example, diffusion of a species that has desorbed in the previous KMC step.
d) Update of values that exist in the event queue. These elements represent events that are affected by a previously occurred reaction, as a result of changing the lateral interactions in the local neighborhood of the just-occurred event. An example of such a case would be updating the inter-arrival time of a desorption event as neighboring sites become more crowded and the spectators exert repulsive forces to the desorbing molecule.

Aiming at improving the timing of the above operations in the context of a KMC simulation, we further developed the skip list data-structure. The insertion operation remains the same as originally proposed.[31] However, the data-structure was heavily adapted in order to improve the efficiency of the deletion operation. The rationale behind the development as well as the algorithm of the modified operations are presented below.

By definition, a skip list enables the execution of the “retrieve minimum value” operation (a) discussed above, since the entries are fully sorted and the element with the minimum key value is located at the very first node, immediately after the head element. We modify the node’s structure so that the unique element identifier, i.e. the label, is also stored within the node in addition to the key value. In the KMC context, this minor modification enables finding both the lattice process label and the time of occurrence in the most efficient way, i.e. in constant time, O(1).

As mentioned further above, deleting an element from a skip list, given its key value, requires searching for it, at a cost of $O(\log_2 N)$. However, if we wish to delete an element that is described by its label rather than its key value, as per operation (c) above, the situation is even worse, since skip list is fully sorted based on key values and not the labels. Searching for a specific label would require accessing all the nodes sequentially, thereby causing the deletion operation to scale as $O(N)$. The need to search elements in the process queue by their label comes from the fact that when a lattice process is executed in *Zacros* (or other KMC implementations), certain subroutines provide the labels of all the lattice processes which are eliminated (becoming obsolete) as a result of the “ongoing” process. Thus, the lattice processes affected by e.g. a diffusional hop, are given by their label, and using this label, one is able to find their occurrence time.

To enable deletion of an element given its label, we created an auxiliary array, called *mapping_array*, which provides access to the skip list's elements by label in ascending order. Each entry of the *mapping_array* is a pointer to the corresponding element of the skip list as shown in Figure 5. It is now possible to retrieve in constant time the element with label *j* (where *j* corresponds to a lattice process) by evaluating *mapping_array[j]*, without going through the whole skip list. For the deletion operation to be fully functional, the skip list is turned from a singly linked-list based into a bidirectional linked-list based. Each node now retains information about its forward nodes but also its backward nodes. Reconnecting the remaining nodes after a deletion is possible using the backwards node instead of the *update_array*. Algorithm 1 presents the core of the deletion operation; looping over the levels of a node and performing reconnections. Through these modifications, the deletion operation is performed in almost constant time, $O(1)$, since only a few node reconnections, with an upper bound of $2\times H_N$, are needed.

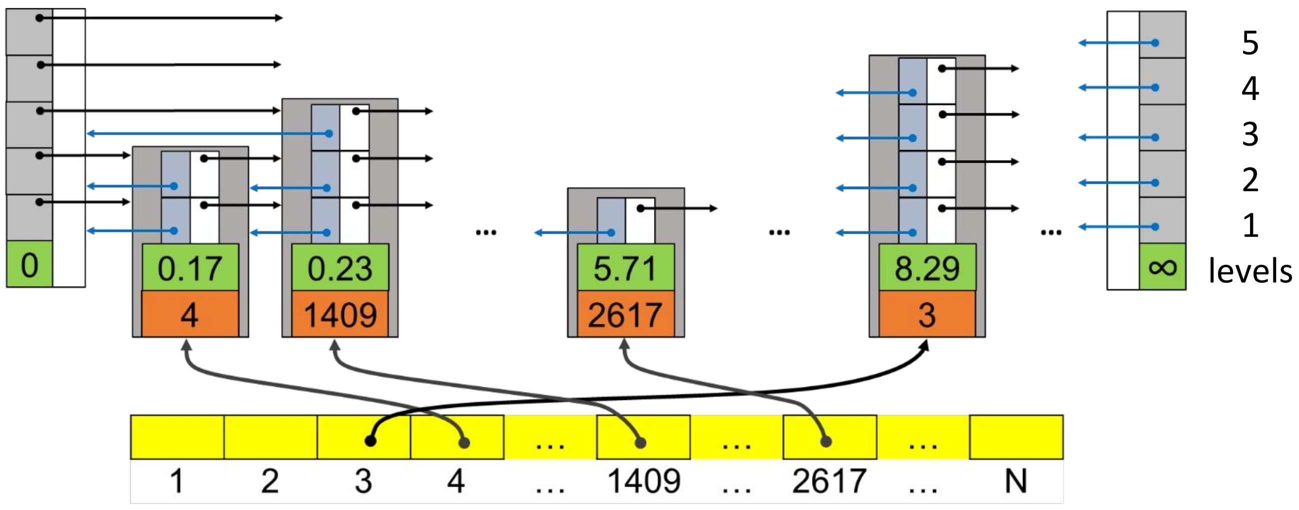


Figure 5: Schematic representation of the developed skip list-based data-structure. Nodes are represented by grey rectangles. The green cells contain the key value while the orange ones contain the corresponding label. The *mapping_array* is represented with yellow color and provides direct access to elements identified by their label. The first (head) and last (tail) node serve as "guard" nodes and they have only forward and backward pointers respectively. Levels are numbered from bottom to top.

Referring to the operation list at the beginning of this subsection, the insertion operation (b) is also adapted according to our modifications. Upon the creation of a new node with key value *t* and label *N+1*, the pointer *mapping_array[N+1]* is connected to the new node. Necessary operations are performed so that the backward pointers of the new node, as well as those of its following node, are connected (refer to Algorithm 2 and Figure 6). It is important to emphasize that all the pointers are pointing to the whole node (illustrated as grey rectangles in Figure 5) and not at an individual level of a certain node. We now proceed to discuss the insertion procedure in more detail with reference to the numbered lines of Algorithm 2 and Figure 6, which represents the operation graphically. Once the new node is created (Alg. 2, 2-4) and its position in the skip list is identified (Alg. 2, 5-6), the next step is to connect it with its predecessor and successor nodes. At first, the new node is linked to its previous node via the *update_array* (Alg. 2, 7.1) which stores the rightmost pointer of every level in the search path. Likewise, the link to its next node is obtained from the *update_array* (Alg. 2, 7.2). Then, the previous node is redirected to point to the new node (Alg. 2, 7.3). Finally, the backward pointer of the next node is redirected to point to the new node as well (Alg. 2, 7.4).

Following the insertion and deletion operations, the update operation, which is a combination of a removal and an insertion, benefits from the improved deletion algorithm. In the following sections, we refer to the developed data-structure described above as *2-way skip list* while we refer to the original skip list-based queueing system by the name *1-way skip list*. These names originate from the direction of the pointers of each data-structure.

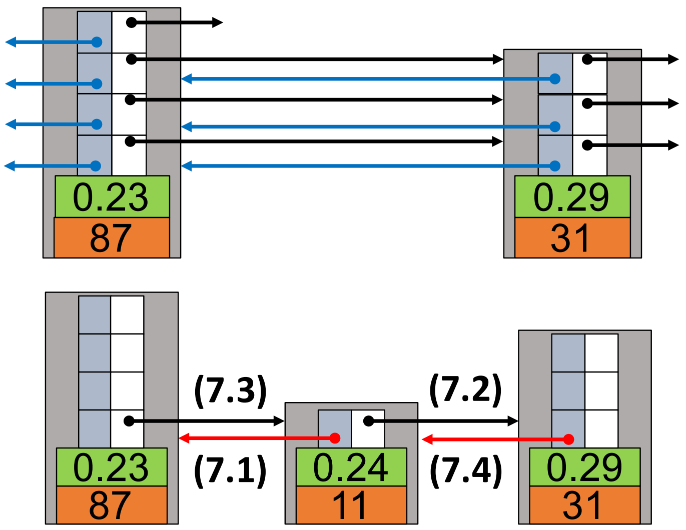


Figure 6: Reconnection of the nodes upon insertion of the element with key value of 0.24 and label 11. Top panel: a part of the queue with the initial connectivity. Bottom panel: connectivity of new pointers after insertion of the new element. The numbered arrows, (7.1)-(7.4), represent the links created by the respective code lines in Algorithm 1. The rest of the pointers are omitted for visual clarity. Green cells correspond to key values; orange cells to labels.

Delete(x)

```
1)    temp → mapping_array[x]
2)    for i:=1 up-to height(temp) do
      2.1)  temp.backward[i].forward[i] → temp.forward[i]
      2.2)  temp.forward[i].backward[i] → temp.backward[i]
```

Algorithm 1: Delete the node with label x.

Insert (list, new_key, new_label)

```
1)    current_list_level := height of the tallest node in list
2)    height := random_height()
3)    new_node → create_node(height, new_key, new_label)
4)    mapping_array[new_label] → new_node
5)    temp → list.head
6)    for i:=current_list_level down-to 1 do
```

```
          6.1)  while temp.forward[i].key < new_key do
                6.1.1)     temp → temp.forward[i]
          6.2)  update_array[i] → temp
7)        for i:=1 up-to height do
          7.1)  new_node.backward[i] → update_array[i]
          7.2)  new_node.forward[i] → update_array[i].forward[i]
          7.3)  new_node.backward[i].forward[i] → new_node
          7.4)  new_node.forward[i].backward[i] → new_node
```

Algorithm 2: Insert a new elementary process to the list with *new_key* and *new_label* as the key and label values respectively.

2.3. Implementation

The data-structures described above were implemented in *Zacros* (a Kinetic Monte Carlo software package written in Fortran 2003 for simulating heterogeneous catalysts), as alternative queueing systems for the *first reaction propagation method*. For further details on the structure of the KMC software package, we refer the reader to Ref. 24,32. Further to the algorithms already presented, a few language-specific additions were incorporated to make the data-structure operational and efficient; nevertheless, the generality of the above algorithms remains unaffected.

3. Benchmark models, results and discussion

In this section, we compare the performance of the queueing systems presented in section 2, which were implemented in KMC software *Zacros*. In order to evaluate the efficiency of these data-structures as queueing systems, benchmark KMC simulations were performed for three simple, yet representative chemical reaction systems: the first two in stationary conditions and the third one in non-stationary conditions. The simulations were performed on an Intel(R) Xeon(TM) E5-1620 processor, running at 3.60 GHz, with 8GB of RAM, under the Ubuntu Linux operating system, version 18.04 LTS. The *Zacros* source code was compiled for serial execution (the OpenMP directives, although present, were not processed) using the GNU Fortran (GFortran) compiler, version 8.3.0.

We would also like to investigate the effect of compiler-induced optimizations on the performance of our queueing systems, because the data-structures we implement differ in their building blocks and compiler optimizations may be able to transform certain operations into a more efficient executable code. In particular, our data-structures are either array based (unsorted list and binary heap) or linked list based (pairing heap, 1-way and 2-way skip list), and compiler optimizations may have different effect on each one of them. Consequently, in order to assess and compare the inherent performance of the various queueing systems, two executables of *Zacros* were compiled, one with compiler optimizations off (option -O0 added during compilation) and the other one with optimizations on (option -O3). Although it is less likely that the unoptimized executable will be useful for production runs, the insights given might be helpful in validation of results, as well as in further development of our software or the compilers. Thus, in the rest of the section, we present the results obtained from both the optimized and unoptimized executables of *Zacros*.

3.1. Stationary simulations

3.1.1. Ziff-Gulari-Barshad model system

The first model used is a continuous-time adaptation of the Ziff-Gulari-Barshad (ZGB) system,[25] as discussed by Stamatakis and Vlachos (Section III.A of Ref. 24). The ZGB is a prototype surface-reaction model for the oxidation of carbon monoxide on a catalytic surface represented by a rectangular lattice. The elementary steps taken into account in the ZGB system are as follows:

$$CO(g) + \ * \rightarrow CO^*$$

$$O_2(g) + \ 2* \rightarrow O^* + O^*$$

$$CO^* + O^* \rightarrow CO_2(g) + 2*$$

where * denotes a vacant site on the catalyst surface, CO* and O* denote adsorbed species, and (g) stands for gas. The partial pressures of CO and O_2 in the gas phase are taken as $P_{CO} = 0.525$ and $P_{O2} = 0.475$ respectively. Since the original algorithm by Ziff et al.[25] operates in discrete time, the kinetic constants were carefully chosen to be appropriate for the continuous time simulation used herein. Summarizing the discussion of Stamatakis and Vlachos,[24] the kinetic constants of the CO adsorption and the O_2 dissociative adsorption are chosen to be proportional to P_{CO} and P_{O2} as $10\cdot P_{CO}$ and $10/4\cdot P_{O2}$ respectively. The constant factor of 10 is arbitrary, and allows for adjusting the frequency of the events per unit time (rescaling of time). Thus, for the aforementioned values, CO adsorption would happen on average $10\cdot P_{CO} = 5.25$ times per KMC time-unit for each empty lattice site. On the other hand, the 1/4 factor in the O_2 dissociative adsorption comes from the 4-fold coordination of each site on the rectangular lattice used for the simulations. Thus, for the given values, O_2 adsorption would happen $10/4\cdot P_{O2}\cdot n_{en} = 1.1875$ times per KMC time-unit for each empty lattice site surrounded by n_{en} empty neighboring sites ($0 \leq n_{en} \leq 4$). The kinetic constant of the CO oxidation and desorption from the surface is taken much larger ($1/4\cdot 10^5$) than the other kinetic constants, so that the instantaneous oxidation assumed by Ziff et al.[25] is reproduced.

3.1.2. Water-gas shift reaction model

The water-gas shift (WGS) reaction produces carbon dioxide and molecular hydrogen from carbon monoxide and water vapour:

$$CO + H_2O \rightarrow CO_2 + H_2$$

It has been the subject of intense theoretical research for the past 30$^+$ years, which has focused on its accurate modelling and on the identification of the rate-limiting step(s) and the optimal reaction conditions.[26,33-37] For our second benchmark, we use a simplified variant of the WGS model on Pt(111) presented in Ref. [26]. The elementary steps taken into account are as follows:

$$CO(g) + \ * \leftrightarrow CO^*$$

$$H_2(g) + \ 2* \leftrightarrow 2H^*$$

$$H_2O(g) + \ * \leftrightarrow H_2O^*$$

$$H_2O^* + \ * \leftrightarrow OH^* + H^*$$

$$OH^* + \ * \leftrightarrow O^* + H^*$$

$$CO^* + OH^* \leftrightarrow COOH^* + \ *$$

$$COOH^* + \ * \rightarrow CO_2(g) + H^* + \ *$$

$$CO^* + O^* \rightarrow CO_2(g) + \ 2*$$

where * denotes a vacant site on the catalyst surface, starred species such as CO*, H*, H_2O* etc. denote adsorbed species, and (g) stands for gas. The partial pressure of the gas species, namely CO, H_2O, H_2 and CO_2, are taken as $P_{CO} = 1.0 \cdot 10^{-8}$, $P_{H2O} = 0.950$ and $P_{H2} = P_{CO2} = 0$ respectively. The procedure for calculating the rate parameters is described in detail in the "Mapping DFT Energies to *Zacros* Input" tutorial webpage of Ref. 38 while the numerical values used in our simulations are provided in the supporting information, section III.

3.1.3. Computational details

The simulations were performed on periodic lattices of increasing size up to a total of about 10^6 lattice sites: starting from 20×20 up to 1000×1000 for the ZGB system, where rectangular lattices were used and from 10×10 up to 707×707 for the WGS model, where hexagonal lattices were used to represent the Pt(111) surface. For each different lattice size, the system being simulated was propagated in time until the surface coverages of the dominant species were fluctuating around constant values, i.e. until stationary behavior was attained. As the dominant species, CO* and O* were chosen for the ZGB model, and CO*, H* and H_2O* for the WGS model, Then, 10^6 KMC steps were further simulated and the simulation clock time needed to execute these 10^6 steps was recorded for post processing. Initially, for every lattice size, five simulations were run, one for each of our five queueing systems. Then, we ran the previous set of five simulations three more times to ensure consistency in our benchmark results. In total, twenty simulations were run for a given lattice size. Worth mentioning, the various queueing systems produced numerically identical results, proving the correct implementation of the data-structures presented as queueing systems for KMC simulations. Lastly, both models presented above do not incorporate lateral interactions between adsorbates, therefore, during the simulations, there were no updates to the occurrence times of the process queue. Consequently, from the queueing system perspective, the update-an-element operation was never invoked. We will refer to this remark later on during the discussion of our results.

3.1.4. Results and discussion

The results from the benchmark simulations appear in Figure 7 where the simulation clock time, viz. the real-time taken to execute the 10^6 KMC steps while on stationary conditions, is plotted against the total lattice sites, N_L. The top panels, Figure 7 (a) and (b), correspond to the runtime scaling of the ZGB system whereas the bottom panels, Figure 7 (c) and (d) correspond to the runtime scaling of the WGS reaction system. Both cases discussed above, with compiler optimizations disabled and enabled are presented in the left (Figure 7 (a) and (c)) and right (Figure 7 (b) and (d)) panels respectively, where each curve represents the average simulation time from four individual runs.

At the first glance, the benchmark results for the ZGB (Figure 7 (a), (b)) and the WGS (Figure 7 (c), (d)) reaction systems appear almost identical. Indeed, the ordering of the queueing systems is largely the same: the computational expense of the unsorted list scales very quickly with the size of the lattice, making this the slowest queueing system followed by the 1-way skip list, the 2-way skip list, the binary heap and the pairing heap. As an exception to these similarities, the binary heap outperforms the pairing heap for the ZBG simulations in the presence of optimizations, as seen in Figure 7 (b). In addition, the differences in the runtimes (and the dissimilar y-axis time ranges) are to be expected, as the runtime is system dependent. As an early conclusion, our results indicate that the performance of our queueing systems is generic regardless of the chemical system under study, as long as the queueing system related dominant operations are the same. We elaborate on the last point in greater depth below. Let us now discuss the performance of each queueing system and compare our results with theoretical expectations.

Referring to Figure 7(a) and (c), we verify the expected poor performance of the unsorted list queueing system due to its linear scaling with system size, $O(\mathrm{N_L})$. Indeed, this queueing system, however simple in implementation, is extremely inefficient in KMC simulations, whereby the find-minimum operation (which is the inefficient one in this queueing system) is performed at every KMC step. For comparison purposes, we note that for the WGS system (Figure 7c), the unsorted list took 9658.6 seconds to complete 10^6 KMC steps on a 707×707 lattice, whereas pairing heap (the fastest queueing system) completed the same simulation in 17.2 seconds and the 1-way skip list in 38.8 seconds. Therefore, the speedups are 562× for the pairing heap and 249× for the 1-way skip list. Crucially, the latter exhibits logarithmic scaling with respect to the system size (compared to the linear scaling of the unsorted list). The developed 2-way skip list has logarithmic scaling as well, but is always faster than its 1-way alternative, especially for larger lattices. This time difference comes from the fact that the deletion operation takes constant time in 2-way skip list, in contrast to $O(\log_2 N_L)$ time in 1-way skip list. Furthermore, the binary heap and pairing heap share the same excellent performance, with the latter being constantly faster than the former by 4% in the ZGB and by 20% - 30% in the WGS. The constant time insertion of pairing heap has a crucial role in the observed performance, since it enables the pairing heap to outperform the array-based binary heap queueing system.

Apart from the unsorted list that exhibits linear scaling, and to which the following discussion does not apply, the other queueing systems exhibit logarithmic scaling with respect to the simulated system size. However, they differ in their multiplicative constant factor. The theoretical time complexities of the performed operations, summarized in Table 1, may clarify the observed performance. In both our benchmark chemical reaction models, only insertions and removals, but no updates, are executed during the simulation. Comparing binary heap and 1-way skip list, we observe that, while both have $O(\log_2 N_L)$ time scaling on the relevant operations, their actual runtime differs substantially. In a binary heap of size N, at every insertion we perform at maximum $\log_2 N$ key value comparisons against the parent of the inserted key value. On the skip list, however, due to its probabilistic nature originating from the random assignment of levels to the nodes, the number of key value comparisons is at most N, the number of elements in the list. Nonetheless, this upper limit is not representative. What is more insightful, instead, is the average number of key value comparisons performed against existing values in the skip list, which is $3/2 \log_2 N + 7/2$, for a skip list with $p=1/2$, as reported by Pugh.[31] These extra key value comparisons are also performed during deletion on the 1-way skip list. Furthermore, the skip list is linked-list based, which means that its nodes are not likely to reside on a consecutive chunk of the computer's memory. On the contrary, binary heap, being array based, occupies memory that is always consecutive, and thus might benefit from lower-level acceleration schemes, such as memory caching. To summarize, the skip list's additional key value comparisons as well as its memory discontinuity incur computational overhead that accumulates over the large number of repetitions and largely contributes to the observed performance.

Queueing system	Find minimum	Insertion	Removal	Update
Unsorted List	O(N)	O(1)	O(1)	O(1)
Binary Heap	O(1)	O($\log_2$N)	O($\log_2$N)	O($\log_2$N)
Pairing Heap	O(1)	O(1)	*O($\log_2$N)*	*O($\log_2$N)*
1-way Skip List	O(1)	*O($\log_2$N)*	*O($\log_2$N)*	*O($\log_2$N)*
2-way Skip List	O(1)	*O($\log_2$N)*	O(1)	*O($\log_2$N)*

Table 1: Expected computational time scaling with respect to the system size, N, of the various operations of all the queueing systems implemented. The expected computational times of pairing heap and both variants of skip list are amortized times (denoted by italics).

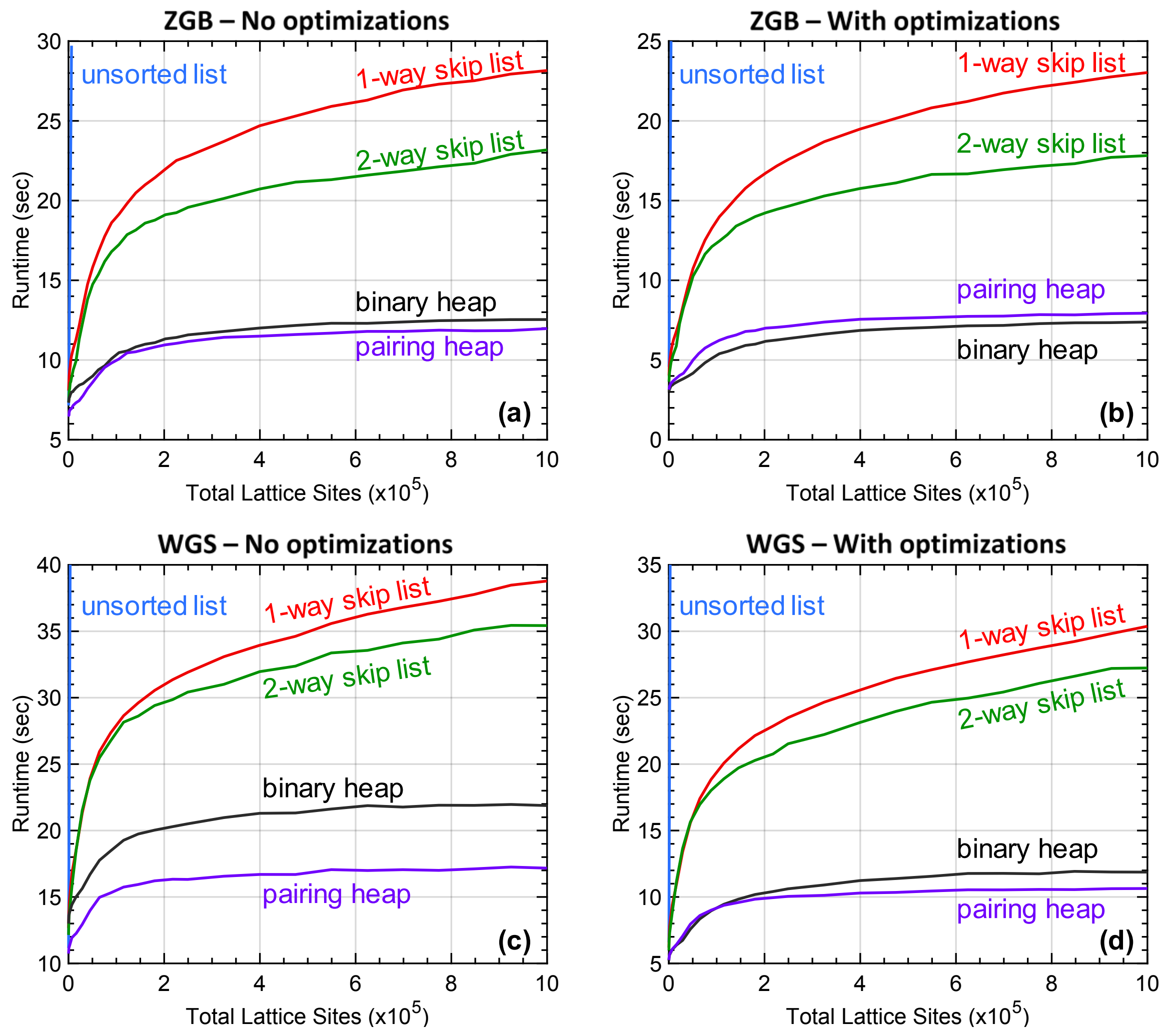


Figure 7: Runtime scaling of the various queueing systems for simulating 10^6 KMC steps while on stationary conditions in the ZGB (top panels) and the WGS (bottom panels) models, with compiler optimizations disabled (left panels) and enabled (right panels).

Considering now the results from the compiler optimized variant, shown in Figure 7(b) and (d) for the ZGB and WGS systems respectively, we observe that the runtime scaling seems similar, apart from two important features. First, the runtime, shown in the y-axis, has dropped significantly for all queueing systems. Second, the binary heap queueing system has outperformed pairing heap in the ZGB system. Let us discuss both in more detail.

Enabling the compiler optimizations had the expected outcome: reduction of the runtime, which implies that the compiler utilized sophisticated techniques and algorithms to execute the same operations more efficiently. The unsorted list benefits from optimizations only when small and moderate size lattices are used. For lattices larger than 100×100 the runtime of the optimized version is almost the same as the unoptimized one. Moving on to the skip list queueing systems, we report that both are, at least, 1.3 times faster than before (i.e. without optimizations) for both the ZGB and WGS models. Lastly, heaps are the most favored, since for the ZGB system, the speed up gain for pairing heap is 1.55×, whereas for the binary heap it is 1.75× (comparing panels (b) and (a) of Figure 7). For the WGS reaction system, the respective speed up gains for the pairing and binary heap are 1.65× and 1.85× (Figure 7 (d) vs (c)). Thus, in the presence of compiler optimizations, comparing the performance of 1-way skip list vs binary heap for the largest lattice size in the ZGB benchmark reveals an acceleration factor of 3.1× (Figure 7 (b)), while for the WGS benchmark, an acceleration factor of 2.8× is obtained for 1-way skip list vs pairing heap (Figure 7 (d)).

The most interesting result from the above benchmarks is the change in the ordering of the pairing heap and binary heap queueing systems, in the ZGB model, when compiler optimizations are enabled. Since no changes were made to our code, we attribute this result, i.e. that binary heap becomes the most efficient queueing system, exclusively to the optimizations. It is very likely that the compiler used in this study (GFortran v8.3.0) is more capable in delivering very efficient machine-level instructions when dealing with array-based data-structures, such as the binary heap. On the other hand, the pairing heap, being a data-structure that heavily relies on pointers, would not benefit from such array-level optimizations. The above explanation is also reinforced by the observed speed up gains as reported in the previous paragraph. In the case of the WGS, the initial separation of the binary and pairing heap curves is substantial, as seen in Figure 7 (c), so that when optimizations are applied, the pairing heap is still the most efficient queueing data-structure, as illustrated in Figure 7 (d).

To summarize, the ZGB and WGS reaction systems used for benchmarks require no updates of the occurrence times due to the absence of lateral interactions. Therefore, only the insertion, find-minimum and removal operations are executed during the simulation. When compiling with no optimizations, the pairing heap queueing system was the most efficient one, followed by the binary heap. As for the optimized version, the binary heap, favored by the compiler optimizations, becomes the most efficient queueing data-structure for the ZGB system, closely followed by pairing heap. In both versions, unoptimized and optimized, and in both benchmark systems, ZGB and WGS, the 2-way skip list is faster than its 1-way variant, showing that our development has had a positive impact on its performance. Nonetheless, the performance improvement was inadequate so that skip list could compete against the binary or the pairing heap.

Reiterating our early conclusions, and based on the above discussion, we note that the qualitative performance of our queueing data-structures is agnostic to the chemical system simulated, in the sense that it only depends on the execution frequency of the insertion, removal and find-minimum operations. In turn, a single execution of these three operations depends on the size of the data-structure holding the occurrence times of all realizable events. In its own turn, the total number of realizable events depends on (a) the total lattice sites and on (b) the number of elementary steps of the chemical model, where the latter is the only information that uniquely identifies the chemical system under study. Information directly related to a chemical system affects the runtime only indirectly via the dependence chain just described. Based on these arguments as well as the similarity of the runtime scaling of the ZGB and the WGS reaction models, it is reasonable to expect that the runtime scaling of the implemented queueing systems will be similar to the results presented above for all other chemical systems simulated, as long as these systems include no lateral interactions.

3.2. Non-stationary simulations

3.2.1. Temperature Programmed Desorption (TPD) model

We will proceed into discussing a set of benchmarks for a different chemical system, but let us first make a brief note on our motivation. The binary heap and pairing heap queueing systems store their elements partially sorted; on the contrary, both skip list variants are fully sorted. Also, the 2-way skip list has a constant time removal operation. In order to make full use of these two properties, we sought a chemical system in which only event deletions happen, as we know that the 2-way skip list is very efficient for this: O(1) vs O($\log_2$N) for all other queueing systems.

To assess the previous rationale, KMC simulations were performed for a Temperature-Programmed Desorption (TPD) model of CO from a pure Cu (111) surface. The only elementary step considered for the TPD model is:

$$CO^* \rightarrow CO(g) \; + \; *$$

The partial pressure of the CO in the gas phase was set to zero in order to simulate ultra-high vacuum environment conditions. Since adsorption events are not taken into account, the reaction mechanism contains only the forward step of the above reaction. The rate constant of CO desorption is calculated from the Arrhenius equation parameterized by DFT calculations, which yield the adsorption energy and the vibrational frequencies for CO on the pure Cu (111) surface (refer to Section 2.3 by Darby et al.).[27] The temperature dependence of the pre-exponential factor of the rate constant is taken into account in the KMC simulation using the following fitted function of T:

$$A_{fwd}(T) = \exp\left[-\left(a_1 \log(T) + \frac{a_2}{T} + a_3 + a_4 T + a_5 T^2 + a_6 T^3 + a_7 T^4\right)\right] \qquad (3)$$

where T is the temperature in K, and α_1 - α_7 are the parameters listed in Table 2. The units of these parameters are such that the exponent of equation (3) is dimensionless and A_{fwd} is evaluated in units of s^{-1}. Lastly, the occurrence times of desorptions are calculated by solving a non-linear version of equation (1) with the Newton-Raphson method; a detailed discussion on how occurrence times are calculated when rate constants are time-dependent is given by Jansen in Ref. 7.

Parameter	Value
α_1	6.535
α_2	-2.762
α_3	-6.300×10^{1}
α_4	-7.076×10^{-2}
α_5	1.885×10^{-4}
α_6	-2.964×10^{-7}
α_7	1.957×10^{-10}

Table 2: Parameter values for fitting the equation (3)

To carry out the TPD simulations and represent the Cu(111) surface, we used hexagonal periodic lattices for which the unit cell contains two sites and each site has a coordination number (number of 1st nearest neighbors), of six. We studied lattices of increasing size, starting from 100×100 up to 1000×1000. At t = 0 s, the temperature is set to 100 K and the whole surface is covered by CO. During the simulation, the temperature ramp was set to 1 K s^{-1} for 100 s to a final temperature of 200 K and the coverage of CO* on the lattice surface was recorded at constant time intervals of 0.05 s. The clock time elapsed for the software to execute the necessary KMC steps was recorded and used as measure of comparison between the queueing systems tested. Similarly to the ZGB and WGS models, all the queueing systems produced numerically identical results, therefore, the difference in computational times is indeed a measure of their efficiency, since all the other operations in the simulation remain the same. At the post-processing stage, the TPD signal was calculated as the moving average of the instantaneous desorption rate, thereby enabling the determination of the temperature at which the CO desorption rate is maximized.

Our benchmark results from the TPD model are shown in Figure 8 and Figure 9 corresponding to the unoptimized and optimized versions respectively. Similar to the benchmarks under stationary conditions, we plot the simulation clock time against the total number of sites on the lattices used. Each curve in Figure 8 and Figure 9 represents the average simulation time from four individual runs. In addition, we normalize the simulation time, namely, we calculate the simulation time per 10^6 KMC steps and plot that quantity against the total lattice sites as shown Figure 10. The unsorted list is excluded from Figure 10 since its computational expense scales very rapidly and is out the range of that of the other queueing systems. The reason we normalize the running time of each queueing system in the TPD model is that the number of adsorbates on the surface is proportional to the KMC steps performed. Therefore, in the smaller lattices, the simulation executes less than 10^6 steps and using only the absolute simulation time (Figure 8 and Figure 9) would prevent us from being able to compare the performance between the stationary (ZGB, WGS) (Figure 7) and non-stationary (TPD) simulations.

A general observation drawn from Figure 8 and Figure 9 is that the scaling of all but the unsorted list queueing systems appears linear with respect to system size, i.e. the number of sites on the lattices; we elaborate on the reason at the end of the current section. It is evident that, in both unoptimized and optimized variants, Figure 8(a) and Figure 9(a) respectively, the unsorted list scales rapidly and becomes extremely inefficient as compared to the other queueing systems. In the unoptimized variant, Figure 8, the two heap-based queueing systems exhibit the same performance. In addition, the two alternatives of the skip list, being faster than binary and pairing heap, require approximately the same time to complete a given simulation. The two-way skip list appear slightly faster than its one-way variant though and the difference in their runtimes increases for larger lattices.

Moving forward to the results in the presence of compiler optimization (Figure 9), we observe that with compiler optimizations enabled and in accordance to our expectations, the performance of the queueing systems is improved. Furthermore, in line with previous observations on the ZGB and WGS systems, the compiler optimizations favor the binary heap over pairing heap. Indeed, both heap-based queueing systems exhibit the same performance before optimizations are applied (Figure 8 (b)), whereas after optimizations (Figure 9 (b)), there is a notable difference between their runtimes, with the binary heap being around 11% faster than the pairing heap. As for the skip list based queueing systems, we observe that their performance is similar and their ordering is preserved, namely, the 2-way skip list is still slightly faster than the 1-way variant. However, for all practical purposes, their performance is identical and both benefit from a 1.5× speed up with respect to their unoptimized counterparts.

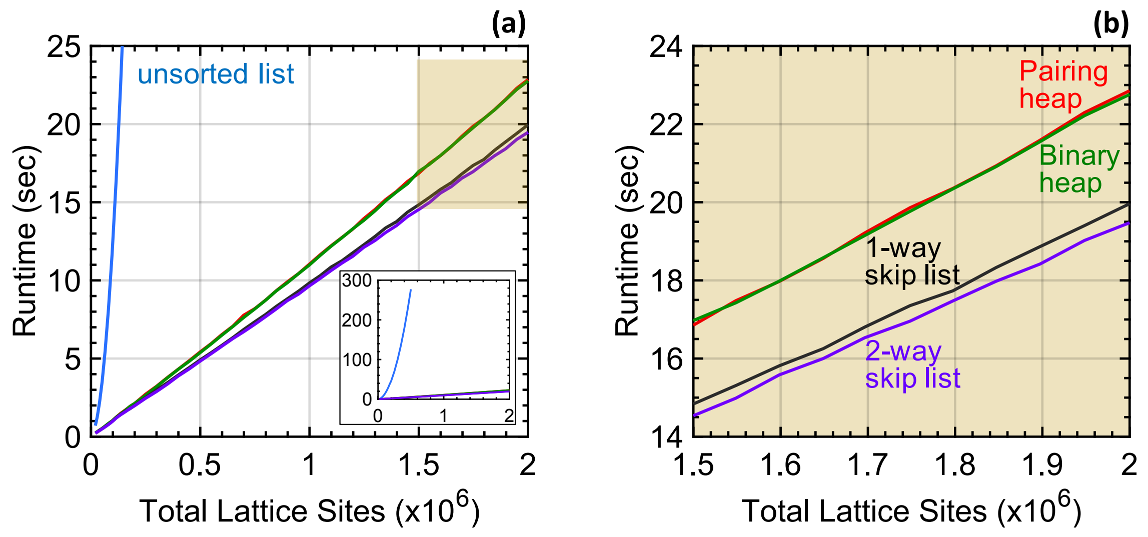


Figure 8: Benchmark results obtained from the unoptimized variant. The inset in **(a)** shows the O(N^2) scaling of the unsorted list; its x- and y-axis represent number of sites and runtime respectively. **(b)** Magnification of the shaded area of (a). The curve labeling in (b) also applies to (a).

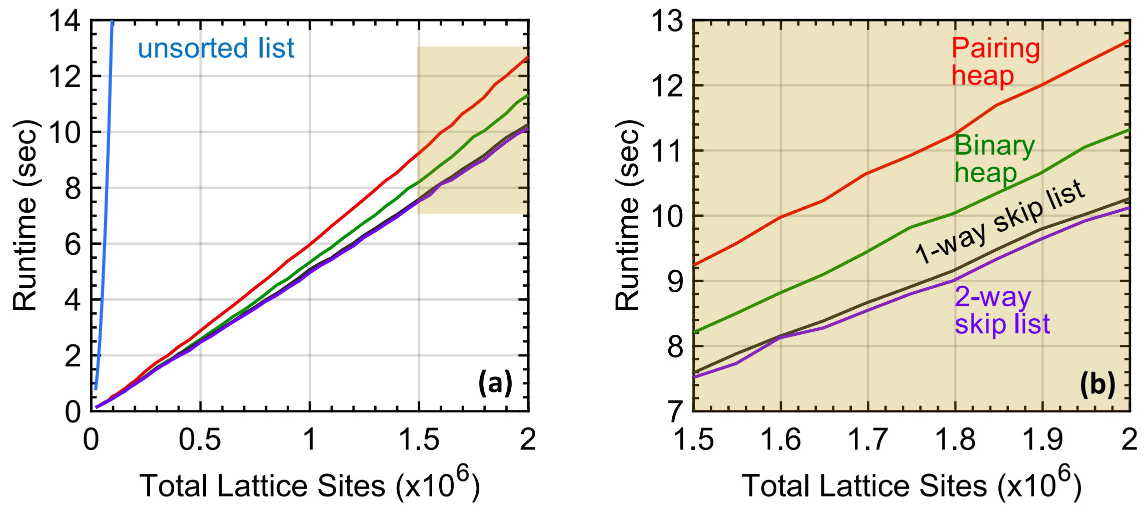


Figure 9: Benchmark results obtained from the compiler-optimized variant of *Zacros*. **(b)** Magnification of the shaded area of (a). The curve labeling in (b) also applies to (a).

The normalized runtime with respect to the total lattice sites, illustrated in Figure 10, also verifies the above conclusions. The normalization of the benchmark results clarifies the behavior of our queueing systems especially in the region of smaller number of lattice sites. It is now clearer that the two-way skip list is consistently faster than the one-way skip list for the unoptimized variant (Figure 10(a)) whereas their difference in runtime becomes very small in the optimized one (Figure 10(b)). Equivalently, pairing heap and binary heap perform the same in the absence of compiler optimizations (Figure 10(a)) whereas they are well separated in the presence of optimizations (Figure 10(b)). In addition, the greater normalized runtime corresponding to small lattices indicates that the bottleneck i.e. the slowest operation of the *Zacros* software is not related to the queueing system. To better understand this, consider a TPD simulation on a fully covered 100×100 hexagonal lattice (20,000 sites). The maximum number of KMC steps that can be simulated is 20,000, equal to the maximum number of adsorbates on the lattice. The absolute simulation time of the above case using the binary heap (or any of the skip list-based) queueing system is 0.130 seconds while the corresponding normalized simulation time is 6.5 seconds. However, since the absolute simulation time obtained is very small, it is not representative of the queueing system because other internal procedures require more time to run and therefore, their contribution on the absolute simulation time dominates. On the other hand, for large lattices, the time required for all other procedures is becoming negligible as compared to the time of the queueing system related operations.

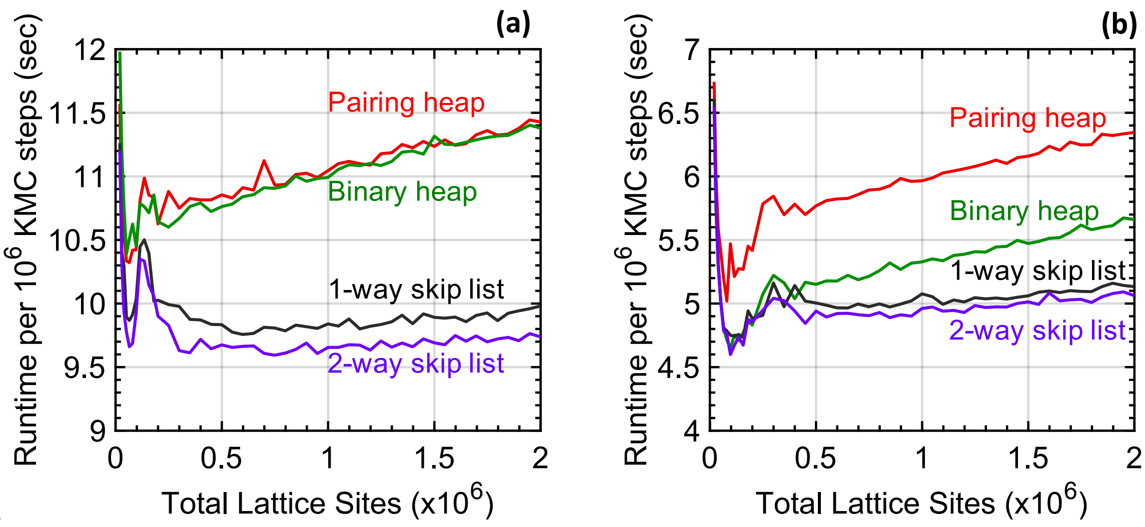


Figure 10: Normalised runtime corresponding to the unoptimized **(a)** and the optimized **(b)** variant of *Zacros*.

For the large lattices, the skip list-based queueing systems outperform the heap-based ones, a completely different outcome as compared to the ZGB and WGS results presented in section 3.1.1 and in particular in Figure 7. The TPD reaction model benefits from the fact that the skip list stores all inter-arrival times fully sorted, the minimum one is located in the beginning of the queue and no other elementary events are added during the simulation. Therefore, the entries in the queue of realizable events are retrieved one after the other, with a small number of pointer reconnections necessary to ensure the queue's proper connectivity. On the contrary, the binary heap stores the inter-arrival times partially sorted. After every event execution, namely finding the minimum and removing it from the queue since the corresponding event has occurred, the algorithm restores the partial order on the heap by performing element swaps (refer to section I of the supplementary information for a schematic representation of the process). In the case where the queue contains a very large number of elements, like the TPD model on large lattices, these frequent swaps incur an overhead on the simulation that accumulates and becomes measurable as shown by the "Binary heap" curves in Figure 10. The same principle applies to the pairing heap as well: following the execution (removal) of an event, the pairing heap has to be rebalanced via operations that are responsible for the observed behavior shown by the curves labelled "Pairing heap" in Figure 10. It is also reasonable to assume that the rebalancing procedure on the pairing heap is not optimized at the same level as the swap procedure on the binary heap. Therefore, the former dominates the runtime and results in the pairing heap having the worst performance as compared to the two skip list-based queueing systems and the binary heap (Figure 9 and Figure 10(b)).

Let us now discuss the observed scaling based on the theoretical time complexity of each queueing system. From the event scheduling perspective, only desorptions occur during the simulation and no other elementary events are added in the queue since adsorption is not considered. Therefore, once the simulation is initiated and every event is inserted in the queue, only the find-minimum and remove-minimum operations are executed at every KMC step. In a lattice with N sites there will be a total of N adsorbates to desorb, hence, N KMC steps are executed. In the unsorted vector the pair of operations "find-minimum"- "remove-minimum" takes O(N) time and is executed N times. Consequently, we expect that the unsorted vector scales quadratically, $O(N^2)$. Indeed, our benchmark results verify the expected scaling (inset in Figure 8(a)). As for the binary and pairing heap queueing systems, the pair of operations mentioned above takes $O(\log_2 N)$ computational time. The simulation terminates after the execution of N desorption events, equivalently after N executions of "find-minimum"-"remove-minimum", and hence the expected time taken is $O(N \log_2 N)$. This scaling behavior mostly resembles the linear scaling (refer to the supplementary information, section III, for a comparison) which is the one we obtained from our benchmarks, in particular, Figure 8(a) and Figure 9(a). Lastly, the "find-minimum"- "remove-minimum" pair of operations takes constant time, O(1) in both skip list variants. That pair is executed N time, therefore, the expected scaling is linear O(N) which is what we observe in both Figure 8(a) and Figure 9(a).

Upon normalization of the runtime, presented in Figure 10, the scaling behavior changes. Binary and pairing heap are expected to scale as $O(\log_2 N)$, whereas in Figure 10 their scaling is mostly linear. It is very likely that we have to go to even larger lattices in order to see the logarithmic scaling, just like in either of the skip list curves in Figure 7(b) where a linearly increasing part precedes the slowly increasing part that compose the entire logarithmic function. As for the skip list-based queueing systems, the expected scaling is O(1), namely constant, since retrieval of the minimum element and its deletion does not depend explicitly on the size of the list. The skip list curves of Figure 10 exhibit a slight increase, however. This is due to the removal operation and more specifically due to the reconnections of the nodes that are bounded by H_N in the one-way skip list and by $2\times H_N$ in the two-way skip list, H_N being the expected height of the skip list. The latter depends on the size of the skip list though (equation (2)) which makes our observation consistent with the theoretical analysis.

The advantage of running TPD simulations in very large lattices is portrayed in Figure 11 where the normalized TPD signal is plotted against the surface temperature. Since the signal is calculated as the derivative of the number of CO molecules desorbed from the surface with respect to time, the greater the number of molecules that desorb per unit time the better the obtained resolution is. Fine resolution is even more important in the presence of multiple desorption peaks, in which case the correct identification of the temperature at which the desorption rate is maximum, depends strongly on the quality of the simulated spectrum.

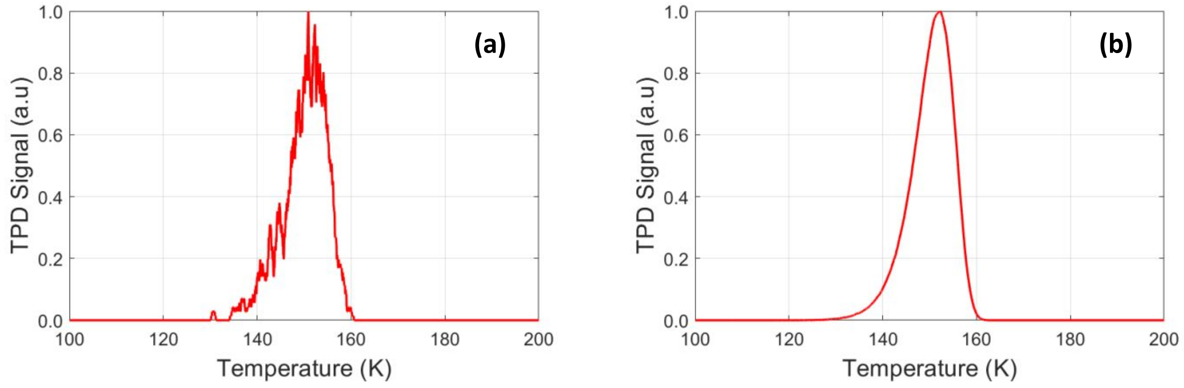


Figure 11: TPD signal from **a)** 20×20 and **b)** 600×600 hexagonal lattice

4. Conclusions

In this study, we compared various data-structures as queueing systems for the *first reaction* method of the KMC algorithm and we further developed a skip list based queueing system. In order to compare their performance in common ground, we implemented these five approaches in the *Zacros* software package and performed benchmark simulations using a CO oxidation model adapted from the ZGB study,[25] a simplified WGS reaction model on Pt(111) based on Ref. 26 and a TPD model of CO on Cu(111).[27] We also studied the effect of compiler optimizations on the computational time required to complete the benchmarks.

The unsorted list was found to be extremely slow, as expected, and thus unusable for production simulations. The binary heap and the pairing heap have the best performance in the ZGB and WGS models, which involve insertion and deletion of elements in the queue. Our data indicates that for such simulations, the heap-based methods could be viewed as equivalent with each other, as their relative efficiency depends on the system and the compiler optimization level. Regarding the skip list-based methods, even though the 2-way skip list consistently appeared to be slightly more efficient in our ZGB and WGS tests, the computational times are quite comparable with each other, so the skip list-based approaches could also be viewed as equivalent. We could potentially be more assertive in stating that the skip list-based queueing systems are less efficient than the heap-based ones, something that makes intuitive sense, since the skip list maintains a fully ordered list of elements, whereas the heap data-structures can only ensure partial ordering. In addition, the heap data-structures, especially the binary heap, appear to be more amenable to compiler optimizations. Still though, skip lists could outperform heap-based queueing systems, e.g. for special cases of systems in which all events are scheduled in the beginning and are executed in that predetermined order. Our TPD model falls precisely into this category and therefore, the skip list queuing systems demonstrated superior performance for these simulations.

The 2-way skip list developed herein is used as a queueing system during a KMC simulation; its usability, however, extends far beyond the applications considered in this work. Any application that involves maintaining a large number of entities in a sorted manner with frequent access to arbitrary elements of this set may take advantage of our proposed design. In particular, choosing to adopt the 2-way skip list would leverage the conditional O(1) element retrieval, the O(logN) generic search and the O(1) element removal operations.

Any *first-reaction* method KMC implementation is, in one way or another, based on the following core operations: event identification and scheduling of realizable events, and execution of the most imminent process. Since KMC is becoming increasingly popular as a tool for understanding and predicting catalytic performance, optimal implementations for each one of the core KMC operations is of paramount importance. In our work, we focused on the efficient event scheduling, thereby evaluating existing data-structures and further developing a novel one (the 2-way skip list) for our purposes. The current work addresses the lack of benchmark studies of not-so-common data-structures for KMC implementations in the scientific literature. Moreover, our benchmarks could serve as a guide for further development of KMC approaches and related software aiming at high efficiency. While these approaches were benchmarked in heterogeneous catalysis problems, they could be applied equally well to other KMC frameworks, simulating for instance diffusion in porous media,[39] or intracellular reactions occurring in biological systems.[9,40]

Supporting Information

Implementation details for the binary and the pairing heap with figures, formation energies and pre-exponential factors for the Water-Gas Shift reaction, comparison of time complexity functions.

This information is available free of charge via the Internet at http://pubs.acs.org

Acknowledgements

This work was partially supported by the Erasmus+ Scheme and the Leverhulme Trust (RPG-2018-209).

References

(1) Rahman, T. S. In *Characterization of Materials*; Kaufmann, E. N., Ed.; Wiley: New York, 2012; pp 253-264.

(2) Voter, A. F. In *Radiation Effects in Solids*; Sickafus, K. E., Kotomin, E. A., Uberuaga, B. P., Eds.; Springer: Dordrecht, 2007; Vol. 235, pp 1-23.

(3) Stamatakis, M. Kinetic modelling of heterogeneous catalytic systems. *J. Phys.: Condens. Matter* **2015,** *27*, 013001.

(4) Andersen, M.; Panosetti, C.; Reuter, K. A practical guide to surface kinetic Monte Carlo simulations. *Front. Chem.* **2019,** *7*, 202.

(5) Reuter, K. In *Modeling and Simulation of Heterogeneous Catalytic Reactions: From the Molecular Process to the Technical System*; Deutschmann, O., Ed.; Wiley-VCH: Weinheim, 2011; pp 71-111.

(6) Darby, M. T.; Piccinin, S.; Stamatakis, M. In *Physics of Surface, Interface and Cluster Catalysis*; Kasai, H., Escaño, M. C. S., Eds.; IOP Publishing Ltd: Bristol, UK, 2016; pp 4.1-4.38.

(7) Jansen, A. P. J. Monte Carlo simulations of chemical reactions on a surface with time-dependent reaction-rate constants. *Comput. Phys. Commun.* **1995,** *86*, 1-12.

(8) Lukkien, J. J.; Segers, J. P. L.; Hilbers, P. A. J.; Gelten, R. J.; Jansen, A. P. J. Efficient Monte Carlo methods for the simulation of catalytic surface reactions. *Phys. Rev. E: Stat. Phys., Plasmas, Fluids, Relat. Interdiscip.Top.* **1998,** *58*, 2598-2610.

(9) Gibson, M. A.; Bruck, J. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. A* **2000,** *104*, 1876-1889.

(10) Chatterjee, A.; Vlachos, D. G. An overview of spatial microscopic and accelerated kinetic Monte Carlo methods. *J. Comput. Aided Mater. Des.* **2007,** *14*, 253-308.

(11) Nielsen, J. H.; Hetherington, J.; Stamatakis, M. ARCHER eCSE01-001 technical report: Zacros software package development: Pushing the frontiers of kinetic Monte Carlo simulation in catalysis. https://www.archer.ac.uk/community/eCSE/eCSE01-001/eCSE01-001_ZACROS_technical_report.pdf (accessed August 26, 2020).

(12) Stamatakis, M.; Ravipati, S.; Christidi, I.; Savva, G. D.; Nielsen, J.; d'Avezac, M.; Guichard, R.; Pineda, M. *Zacros: Advanced lattice-KMC made easy*, Revision 3.0dev; London, UK, 2020.

(13) Plimpton, S.; Battaile, C.; Chandross, M.; Holm, L.; Thompson, A.; Tikare, V.; Wagner, G.; Webb, E.; Zhou, X.; Cardona, C. G. Crossing the mesoscale no-man's land via parallel kinetic Monte Carlo. *Sandia Report SAND2009-6226* **2009,** *1*.

(14) Plimpton, S.; Thompson, A.; Slepoy, A. SPPARKS kinetic Monte Carlo simulator. http://spparks.sandia.gov/ (accessed June 17, 2020).

(15) Leetmaa, M.; Skorodumova, N. V. KMCLib: A general framework for lattice kinetic Monte Carlo (KMC) simulations. *Comput. Phys. Commun.* **2014,** *185*, 2340-2349.

(16) Hoffmann, M. J.; Matera, S.; Reuter, K. kmos: A lattice kinetic Monte Carlo framework. *Comput. Phys. Commun.* **2014,** *185*, 2138-2150.

(17) Chill, S. T.; Welborn, M.; Terrell, R.; Zhang, L.; Berthet, J. C.; Pedersen, A.; Jonsson, H.; Henkelman, G. EON: software for long time simulations of atomic scale systems. *Modell. Simul. Mater. Sci. Eng.* **2014,** *22*, 055002.

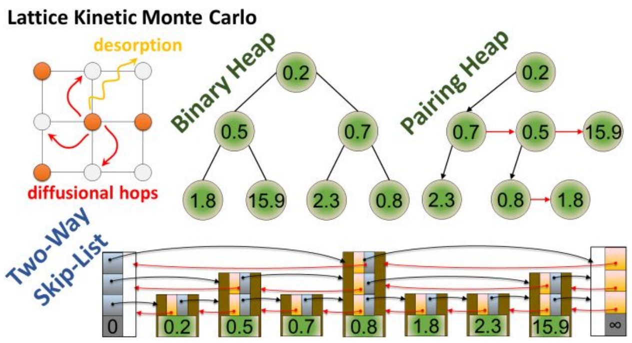
(18) Lukkien, J. J.; Jansen, A. P. J. CARLOS Project: a general purpose program for the simulation of chemical reactions taking place at crystal surfaces. http://carlos.win.tue.nl/ (accessed June 17, 2020).

(19) Jorgensen, M.; Gronbeck, H. MonteCoffee: A programmable kinetic Monte Carlo framework. *J. Chem. Phys.* **2018,** *149*, 114101.

(20) Kunz, L.; Kuhn, F. M.; Deutschmann, O. Kinetic Monte Carlo simulations of surface reactions on supported nanoparticles: A novel approach and computer code. *J. Chem. Phys.* **2015,** *143*, 044108.

(21) Ramsey, S.; Orrell, D.; Bolouri, H. Dizzy: stochastic simulation of large-scale genetic regulatory networks. *J. Bioinf. Comput. Biol.* **2005,** *3*, 415-436.
(22) Maarleveld, T. R.; Olivier, B. G.; Bruggeman, F. J. StochPy: A comprehensive, user-friendly tool for simulating stochastic biological processes. *PLoS One* **2013,** *8*, e79345.
(23) Kazeroonian, A.; Frohlich, F.; Raue, A.; Theis, F. J.; Hasenauer, J. CERENA: ChEmical REaction Network Analyzer-A toolbox for the simulation and analysis of stochastic chemical kinetics. *PLoS One* **2016,** *11*, e0146732.
(24) Stamatakis, M.; Vlachos, D. G. A graph-theoretical kinetic Monte Carlo framework for on-lattice chemical kinetics. *J. Chem. Phys.* **2011,** *134*, 214115.
(25) Ziff, R. M.; Gulari, E.; Barshad, Y. Kinetic phase-transitions in an irreversible surface-reaction model. *Phys. Rev. Lett.* **1986,** *56*, 2553-2556.
(26) Stamatakis, M.; Chen, Y.; Vlachos, D. G. First-principles-based kinetic Monte Carlo simulation of the structure sensitivity of the water-gas shift reaction on platinum surfaces. *J. Phys. Chem. C* **2011,** *115*, 24750-24762.
(27) Darby, M. T.; Sykes, E. C. H.; Michaelides, A.; Stamatakis, M. Carbon monoxide poisoning resistance and structural stability of single atom alloys. *Top. Catal.* **2018,** *61*, 428-438.
(28) Gillespie, D. T. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *J. Comput. Phys.* **1976,** *22*, 403-434.
(29) Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. *Introduction to Algorithms, Third Edition*; The MIT Press: 2009.
(30) Fredman, M. L.; Sedgewick, R.; Sleator, D. D.; Tarjan, R. E. The pairing heap: A new form of self-adjusting heap. *Algorithmica* **1986,** *1*, 111-129.
(31) Pugh, W. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM* **1990,** *33*, 668-676.
(32) Nielsen, J.; d'Avezac, M.; Hetherington, J.; Stamatakis, M. Parallel kinetic Monte Carlo simulation framework incorporating accurate models of adsorbate lateral interactions. *J. Chem. Phys.* **2013,** *139*, 224706.
(33) Ovesen, C. V.; Stoltze, P.; Norskov, J. K.; Campbell, C. T. A kinetic-model of the water gas shift reaction. *J. Catal.* **1992,** *134*, 445-468.
(34) Grabow, L. C.; Gokhale, A. A.; Evans, S. T.; Dumesic, J. A.; Mavrikakis, M. Mechanism of the water gas shift reaction on Pt: First principles, experiments, and microkinetic modeling. *J. Phys. Chem. C* **2008,** *112*, 4608-4617.
(35) Ratnasamy, C.; Wagner, J. P. Water gas shift catalysis. *Cat. Rev. - Sci. Eng.* **2009,** *51*, 325-440.
(36) Prats, H.; Alvarez, L.; Illas, F.; Sayos, R. Kinetic Monte Carlo simulations of the water gas shift reaction on Cu (111) from density functional theory based calculations. *J. Catal.* **2016,** *333*, 217-226.
(37) Yang, M.; Flytzani-Stephanopoulos, M. Design of single-atom metal catalysts on various supports for the low-temperature water-gas shift reaction. *Catal. Today* **2017,** *298*, 216-225.
(38) Stamatakis, M.; Ravipati, S.; Christidi, I.; Savva, G. D.; Nielsen, J.; d'Avezac, M.; Guichard, R.; Pineda, M. Zacros: Advanced lattice-KMC made easy. http://zacros.org/ (accessed June 17, 2020).
(39) Apostolopoulou, M.; Santos, M. S.; Hamza, M.; Bui, T.; Economou, I. G.; Stamatakis, M.; Striolo, A. Quantifying pore width effects on diffusivity via a novel 3D stochastic approach with input from atomistic molecular dynamics simulations. *J. Chem. Theory Comput.* **2019,** *15*, 6907-6922.
(40) Stamatakis, M.; Zygourakis, K. A mathematical and computational approach for integrating the major sources of cell population heterogeneity. *J. Theor. Biol.* **2010,** *266*, 41-61.

TOC Graphic


Lattice Kinetic Monte Carlo
desorption
diffusional hops
Binary Heap
0.2
0.5
0.7
1.8
15.9
2.3
0.8
Pairing Heap
0.2
0.7
0.5
15.9
2.3
0.8
1.8
Two-Way Skip-List
0
0.2
0.5
0.7
0.8
1.8
2.3
15.9
∞

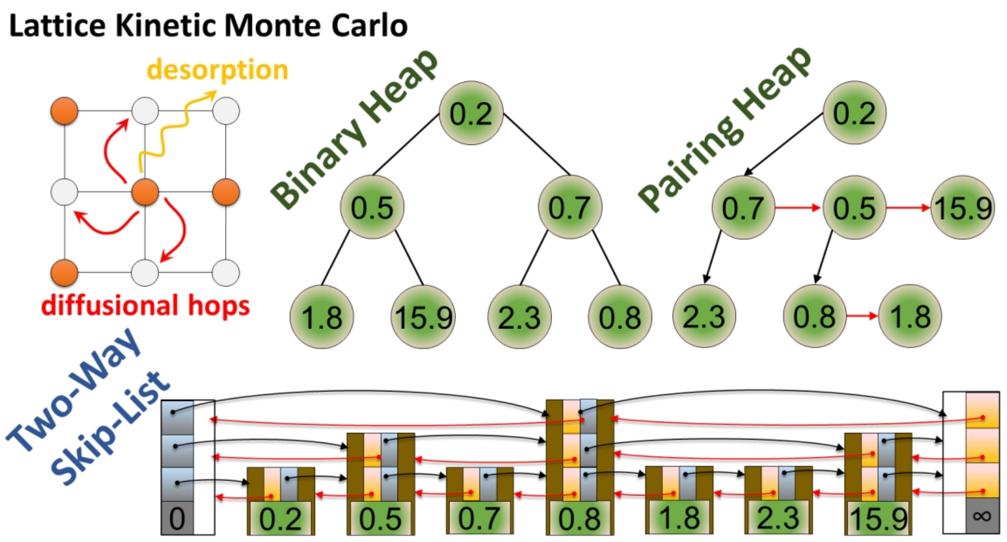


Table of Contents graphic

445x239mm (96 x 96 DPI)