A FAST EXACT SIMULATION ALGORITHM FOR A CLASS OF MARKOV JUMP PROCESSES

YAO LI AND LILI HU

ABSTRACT. A new stochastic simulation algorithm, named the Hashing-Leaping (HL) algorithm, for exact simulations of a class of Markov jump processes, is presented in this paper. The HL algorithm has constant computational cost per event, which is independent of the number of exponential clocks in the Markov process. The main idea of the HL algorithm is to repeatedly implement a Hash-table-like bucket sort algorithm for all times of occurrence covered by a time step with length $\tau$. This paper serves as an introduction to this new algorithm. We introduce the algorithm, demonstrate its implementation, and compare its performance with the benchmark NRM algorithm in three examples. Our performance test and CPU operation statistics show a significant advantage of the HL algorithm for large scale problems.

1. Introduction

Since the late 1960s, much effort has been devoted to simulations of Markov jump processes on high dimensional state spaces. Most of these Markov jump processes arise in two classes of problems. The first is usually called biochemical reaction networks. It is well accepted that when the number of species is small, due to stochastic effects, a deterministic differential equation fails to model real-world chemical reactions accurately. Therefore, numerous chemical reaction systems within biological cells, such as gene networks, regulatory networks, and signaling pathways networks, are modeled by Markov jump processes. The second type of problems are related to the kinetic Monte Carlo (KMC) method [29], which essentially covers all stochastic evolution models that proceed as a sequence of infrequent transitions at heterogeneous, state-dependent exponential random times. The kinetic Monte Carlo method was first introduced to simulate the radiation damage [2]. Today, it is used to generate stochastic trajectories appearing in communication networks, factory scheduling, surface/crystal growth, and chemical/physical vapor deposition [1][2][21][29].

Markov jump processes coming from both KMC and chemical reaction networks have some common features. They are all driven by finite many independent exponential clocks. The state of the Markov process will be updated when a clock rings. The rate of each clock depends on the current state of the process. Therefore the update follows after the ringing of an exponential clock can change the rate of other exponential clocks. In most applications, the update is a simple transformation, under which the rates of most clocks remain unchanged. It is worth to mention that

2010 Mathematics Subject Classification. Primary: 60J22, 65C05, 65C40.
besides chemical reaction and KMC, a variety of Markov jump processes in statistical physics, such as the kinetic Ising model, the simple inclusion process (SIP), the simple exclusion process (SEP), and many of their variants like TASEP, ASEP etc, can also be categorized into this family.

The scale of these Markov jump processes can be very large. For example, some chemical reaction networks have thousands of reactions, while the scale of some reaction-diffusion equations can be as large as several millions. Therefore, it is important to design fast algorithms for those large scale problems. As a Markov jump process proceeds sequentially at a series of state-dependent random times, the fundamental rule of an exact simulation is to always find the next occurring event, and to update exactly one event at a time. Early developed algorithms like the Gillespie direct method (DM) [17], the first reaction method (FRM) [16], and the BKL algorithm [5] for KMC rely on a linear search of times of occurrence of events. More sophisticated algorithms, like the next reaction method (NRM) and the composition-rejection method (CR), use a binary heap or other mechanisms to identify the next occurring event [15,27]. The performance of those simulation algorithms is usually measured by computational cost per event. Let $N$ be the number of exponential random times in a Markov jump process. Then the computational costs of per event are $O(N)$ for early algorithms like DM, FRM, and BKL, $O(\log N)$ for recent algorithms like the next reaction method (NRM), and conditional $O(1)$ for some newly introduced algorithms like the CR algorithm. Besides the exact simulation algorithms mentioned above, there are also approximate algorithms such as the tau-leaping algorithm and its numerous variants [7,9,18].

A large family of enhanced algorithms are also developed specifically for problems in different applications [6,8,10,25,26,30,31]. Important algorithms that are worth to mention include: the multi-scale stochastic simulation algorithm (MSSA) for chemical reaction networks with multiple time scales [6,30], the optimized direct method (ODM) for exact simulations of chemical reaction networks with heterogeneous reaction rates [8], and the next subvolume method (NSM) for stochastic reaction-diffusion systems [14].

The aim of the present paper is to introduce the Hashing-Leaping (HL) algorithm, which is an exact simulation algorithm for Markov jump processes with $O(1)$ per event computational complexity. The mechanism of this algorithm is motivated by the bucket sort algorithm. We repeatedly leap forward the time by a constant $\tau$, then use a Hash-table-like algorithm to distribute random times covered by the leaping step into $Q$ buckets. By choosing suitable $Q$ and $\tau$, the average number of time of occurrence in each bucket can be controlled at $O(1)$. Then we will sequentially update all events in each bucket until the next leaping step. This algorithm is very easy to implement. With proper choice of parameters, it is faster than all the other algorithms we have tested. In addition, as will be explained in Section 3, the Hashing-Leaping algorithm is compatible to parallel machine, especially for Markov jump processes with sparse dependency graphs. This paper serves as an introduction to the HL algorithm. We plan to investigate subtle issues like the
SIMULATION ALGORITHM FOR MARKOV JUMP PROCESSES

choice of parameters, further enhancements of the algorithm, and parallelizations in our subsequent works.

The organization of this paper is as follows. In Section 2, we will describe the family of Markov jump processes that we are interested in, which covers Markov processes derived from chemical reaction networks and kinetic Monte Carlo simulations. Section 3 presents a short review of mainstream stochastic simulation algorithms. The HL algorithm is introduced in Section 4. Three numerical examples, including a generalized KMP model, a chemical reaction network, and a reaction-diffusion system, are presented in section 5. Section 6 is the conclusion.

2. Description of the Markov jump process model

We first give a generic description of the Markov jump process to be studied in the present paper. Let \( X_t = \{x^t_i\}_{i=1}^M \) be a Markov jump process on \( \mathbb{R}^M \) that is determined by \( N \) random times, which are generated by mutually independent exponential clocks. The rates of those clocks are state-dependent, denoted by \( R_1(X_t), \ldots, R_N(X_t) \), respectively, where \( R_i : \mathbb{R}^M \to \mathbb{R}^+ \) are called rate functions. An update transformation \( T^{\omega_i}_i : \mathbb{R}^M \to \mathbb{R}^M \) is associated to each exponential clock, where \( \omega_i \in \Omega_i \) is a random parameter whose probability measure is \( p_i(d\omega_i) \), and \( \Omega_i \) is the sample space of \( p_i \). When the \( i \)-th clock rings, called an “event”, \( X_t \) is updated by a random transformation \( T^{\omega_i}_i \). During the update, the random parameter \( \omega_i \) is chosen from the probability measure \( p_i \), independent of everything else. After the update, \( X_t \) jumps to a new state \( X_{t+} = T^{\omega_i}_i(X_t) \). Throughout the present paper, unless specified otherwise, \( N \) means the number of exponential clocks, which is essentially the scale of the Markov jump process.

The dependency graph of \( X_t \) is a directed graph \( G = (V, E) \) with \( N \) vertices representing \( N \) exponential clocks. \( \{i, j\} \) forms a directed edge if and only if \( R_j \neq R_j \circ T_i \). In other words, \( (i, j) \) is an edge if and only if the \( i \)-th clock affects the \( j \)-th clock.

It is easy to check that Markov jump processes arise in a very large family of applied problems, including statistical mechanics, chemical reaction networks and kinetic Monte Carlo simulations, fit the description of \( X_t \). In those applications, \( M \) and \( N \) can be very large numbers but \( R_i \) only depends on a limited number of coordinates of \( X_t \), and \( T_i \) is an identical transformation on all but a limited number of coordinates. Therefore, the dependency graph is usually very sparse comparing with the scale of \( M \) or \( N \). For example, in a stochastic chemical reaction network \( X_t = \{x^t_i\}_{i=1}^M \) represent the number of molecules of \( M \) reactant species, and \( \{R_i\}_{i=1}^M \) represent the rate of each reaction, or the “propensity”, which is determined by a monomial of several \( x_i \)s, and \( T_i(X_t) = X_t + v_i \). The vector \( v_i \in \mathbb{R}^N \) is a sparse vector with all zero entries except those corresponding to reactant species involved in the \( i \)-th reaction.

3. A short review of existing algorithms

3.1. Direct method. As explained in the introduction, when simulating \( X_t \), it is important to note that those \( M \) clocks are mutually independent on a time interval
only if all rates $R_i$ remain unchanged. When one clock rings, the corresponding update transformation $T_i$ will change the rates of other clocks. With a small but positive probability, this will lead to a “chain reaction” of events and change the rates of all clocks in a short time frame. Therefore, to simulate $X_t$, it is crucial to always identify the “next event”.

The first two popular algorithms of simulating Markov jump processes like $X_t$ were developed by Gillespie [16][17], called the direct method (DM) and the first reaction method (FRM), respectively. In the literature of kinetic Monte Carlo simulations, DM is also called the BKL algorithm, which was developed independently by the other group of researchers [3]. In DM, two random variables are generated to select each event. The first random variable determines the time of occurrence of the next event, and the second one is used to select the event together with a linear search. In FRM, a linear search is used to select both the time of occurrence and the event that will occur next at the same time. Due to the linear search, the computational costs of both algorithms are $O(N)$ per event. The algorithms can be summarized as follow.

**Direct Methods**

**Step 1:** Initialize $R_i$s, let $R_{\text{sum}} = \sum_{i=1}^{N} R_i$ and $T = 0$

**Step 2:** Generate an exponential random variable with rate $R_{\text{sum}}$, denoted by $\Delta t$

**Step 3:** Let $T = T + \Delta t$. Generate a uniform random variable on $[0, 1]$, denoted by $p$

**Step 4:** Find the minimum $l$ such that $pR_{\text{sum}} < \sum_{i=1}^{l} R_i$

**Step 5:** Make update according to $T_l$

**Step 6:** Recalculate all rate functions and $R_{\text{sum}}$

**Step 7:** Return to step 2 or quit

**First Reaction Method**

**Step 1:** Initialize $R_i$s

**Step 2:** Generate $N$ exponential random variables $t_1, \ldots, t_N$ with rates $R_1, \ldots, R_N$, respectively.

**Step 3:** Use linear search to find the least time, denoted by $t_l$

**Step 4:** Make update according to $T_l$

**Step 5:** Recalculate all rate functions

**Step 6:** Return to step 2 or quit

It is a standard exercise to show that these two algorithms are equivalent. Gillespie direct method has many variants, such as the optimized direct method (ODM) introduced by Cao et al [8].

3.2. **Next reaction method.** FRM was significantly optimized by Gibson and Bruck [15], called the next reaction method (NRM). Main improvements of NRM include:

- Introduce the concept of dependency graphs. Only rate functions affected by an event will be updated
- Reuse times of occurrence of events without regenerating random variables
- Use a (minimum) binary heap to reduce the search time to $O(1)$ and the update time to $O(\log N)$
NRM reduces the average computational cost per event to \(O(\log N)\). Currently, this algorithm is widely used in stochastic simulation packages and commercial softwares. We also choose NRM as the benchmark algorithm to test the performance of our algorithm.

The next reaction method can be summarized as follows

**Step 1:** *Initialization:*
(a) Initialize \(R_i\)s
(b) Construct the dependency graph \(G\)
(c) Generate \(N\) exponential random variables \(t_1, \ldots, t_N\) with rates \(R_1, \ldots, R_N\), respectively
(d) Store times of occurrence into a binary heap

**Step 2:** Find the event on the top of the binary heap, denoted by \(t_1\)

**Step 3:** Make update according to \(T_1\)

**Step 4:** Follow the dependency graph to update all affected rate function \(R_i\)s

**Step 5:** Update times of occurrence of all affected events as
\[
t_{i}^{\text{new}} = (t_{i}^{\text{old}} - t_1) \cdot \frac{R_i}{R_{i}^{\text{old}}} + t_1	ag{*}
\]
and maintain the binary heap

**Step 6:** Generate a new \(t_1\) and maintain the binary heap

**Step 7:** Go to step 2 or quit.

### 3.3. Composition-Rejection method.

The idea of the composition-rejection method (CR) \[27\] comes from a simple probabilistic fact used in the Gillespie direct method. If we have \(N\) mutually independent exponential random variables \(Y_1, \ldots, Y_N\) with rates \(r_1, \ldots, r_N\), respectively. Let \(Y_{\text{min}}\) be the minimum of the \(N\) random variables. Let \(r_{\text{sum}}\) be the sum of the \(N\) rates. Then \(Y_{\text{min}}\) is an exponential random variable with rate \(r_{\text{sum}}\). In addition,
\[
\mathbb{P}[Y_{\text{min}} = Y_i] = \frac{r_i}{r_{\text{sum}}}.
\]

Therefore, sampling \(Y_i = Y_{\text{min}}\) is equivalent to sampling a weighted distribution on \(\{1, \ldots, N\}\) with weights \(r_1/r_{\text{sum}}, \ldots, r_N/r_{\text{sum}}\), respectively. Instead of the linear search as in direct method, this sampling can also be done by the following rejection-based method. Let \(r_{\text{max}} = \max\{r_1, \ldots, r_N\}\). Two random numbers \(Z_1\) and \(Z_2\) are repeated generated until \(Y_{\text{min}}\) is selected, where \(Z_1\) is uniformly distributed on \([0, r_{\text{max}}]\), and \(Z_2\) is an integer that is uniformly distributed between 1 and \(N\). The rule of the rejection is that: if \(Z_1 > r_{Z_2}\), the pair \((Z_1, Z_2)\) is rejected, otherwise it is accepted and we have \(Y_{\text{min}} = Y_{Z_2}\).

The rejection-based sampling method has constant computational complexity independent of \(N\). However the constant can be very large if \(r_{\text{max}}\) is much larger than all the other \(r_i\)s. This is partially solved by the composition-rejection method. The CR method relies on the maximal and minimal rates \(R_{\text{min}}\) and \(R_{\text{max}}\). To reduce the expected number of rejections, \(R_1, \ldots, R_N\) are distributed into \(g = \lceil \log_2(R_{\text{max}}/R_{\text{min}}) \rceil\) groups. The first group contains rate functions ranging from \(R_{\text{min}}\) to \(2R_{\text{min}}\), the second group from \(2R_{\text{min}}\) to \(4R_{\text{min}}\), and so on. According
to \([27]\), \(g \leq 30\) is sufficient for most applications. Sums of rates in each groups are calculated, denoted by \(p_1, \ldots, p_g\), and stored. The CR method uses Gillespie direct method to select the group (composition), and uses the rejection-based sampling technique introduced above to select the event (rejection) in the selected group. After updating an event, \(R_{\text{min}}, R_{\text{max}}\) and all affected groups will be maintained.

Composition-rejection method can reduce the computational cost per event for the Markov jump processes with very large number of clocks. According to the performance test in \([27]\) for a chemical reaction network, the composition-rejection method is \(2 \sim 3\) times faster than NRM when \(N\) ranges from \(1 \times 10^6\) to \(1 \times 10^9\).

The composition-rejection method can be summarized as follows:

**Step 1:** Initialization:
(a) Initialize \(R_i\)s, let \(T = 0\)
(b) Construct the dependency graph \(G\)
(c) Distribute \(N\) clocks into \(g\) groups and compute \(p_1, \ldots, p_g\), the sum of rates in each groups
(d) Calculate \(R_{\text{sum}} = \sum_{i=1}^{N} R_i\)

**Step 2:** Generate an exponential random variable with rate \(R_{\text{sum}}\), denoted by \(\Delta t\)

**Step 3:** Let \(T = T + \Delta t\). Generate a uniform random variable on \([0, 1]\), denoted by \(u\)

**Step 4:** Find the minimum \(k\) such that \(uR_{\text{sum}} < \sum_{i=1}^{k} p_i\)

**Step 5:** Use rejection-based sampling method to select an event \(R_l\) from group \(p_k\)

**Step 6:** Make update according to \(T_l\)

**Step 7:** Follow the dependency graph to update all affected rate function \(R_i\)s

**Step 8:** Update times of occurrence of all affected events as
\[
t_{i}^{\text{new}} = (t_{i}^{\text{old}} - t_l) \cdot \frac{R_{i}^{\text{old}}}{R_{i}^{\text{new}}} + t_l
\]

**Step 9:** Generate a new \(t_l\)

**Step 10:** Maintain the groups, update \(R_{\text{sum}}\) and affected \(p_i\)s

**Step 11:** Go to step 2 or quit.

### 4. Hashing-Leaping (HL) Algorithm

#### 4.1. Introduction to the algorithm
We will introduce a constant time per event algorithm for the exact simulation of \(X_t\). As reviewed in the last section, the main bottleneck of simulating \(X_t\) is selecting the event that occurs next. Instead of a linear search or a heap sort, our algorithm is motivated by the bucket sort algorithm, which is an \(O(N)\) complexity sort algorithm in most practical settings. To simulate \(X_t\), two parameters \(\tau\) and \(Q\) are chosen from either observing rate function \(R_i\)s or smaller scale simulations, where \(\tau > 0\) is the step size and the positive integer \(Q\) is the number of buckets.

The algorithm runs in the following way: The times of occurrence of events associated to \(N\) clocks, denoted by \(t_1, \ldots, t_N\), are recorded and maintained. The algorithm makes a major update in the beginning of every time step with length \(\tau\),
called a bucket redistribution. In the $n$-th bucket redistribution, $t_1, \cdots, t_N$ are distributed into $Q+1$ buckets, denoted by $B_1, \cdots, B_Q, B_L$, that represent time intervals $(n-1)\tau \leq t < (n-1)\tau + \tau/Q$, $(n-1)\tau + \tau/Q \leq t < (n-1)\tau + 2\tau/Q, \cdots, (n-1)\tau + (Q-1)\tau/Q \leq t < nt$, and $t > nt$, respectively. Then we start from the first non-empty bucket $B_{n_1}$, find the minimum time of occurrence, say $t_k$, by a linear search, and make update according to $T_k$. During the update, the following two operations will be carried out: (1) An exponential random variable with rate $R_k$ will be generated and added to $t_k$, which is the time of occurrence of the next event associated to $R_k$. $t_k$ will then be placed into the proper bucket. (2) We then follow the dependency graph to update the new clock rates of all affected clocks, as well as the corresponding times of occurrence. It is a standard knowledge that affected ringing times can be changed as in equation (1) without generating a new random number (Theorem 2 in [15]), as used in NRM and CR. The reason is that the transformation in equation (1) maps an exponential distribution with rate $R_i^{old}$ to an exponential distribution with rate $R_i^{new}$. The updated times of occurrence of events will be placed into the proper buckets. The same procedure will be carried out repeatedly until the first bucket becomes empty.

Then we move to the next nonempty bucket and carry out the same series of operations until it becomes empty. This procedure continues until all buckets $B_1, \cdots, B_Q$ are emptied. At that time, times of occurrence of all events are stored in $B_L$. Then the information stored in the bucket $B_L$ will be used to perform the next bucket redistribution. We call it the Hashing-Leaping algorithm because the bucket redistribution step resembles the Hash algorithm, while the whole algorithm can be seen as an exact version of the tau-leaping algorithm.

The HL algorithm should be implemented with proper data structure to improve the efficiency. The simplest way we find is to construct an array of $N$ structs, named TimeArray. Each struct, of the type ST, has three elements: a floating number that indicates the time of occurrence of the event associated to the exponential clock, a left pointer and a right pointer, both of the type ST. In addition we need an array of $Q+1$ pointers of the type ST, called BucketArray, that represents the heads of $Q+1$ buckets. A floating number array RateArray is also needed to store the rate of each clock.

After a bucket redistribution, each bucket is formed by a doubly linked list whose head is pointed by an element in BucketArray, as shown in Figure 1. When updating each bucket, a linear search is performed to find the minimum time of occurrence within this bucket. Every update of a time of occurrence requires two operations: (1) removing the corresponding ST struct from its old bucket, i.e., setting its left and right pointers as NULL and maintain the doubly linked list; and (2) pushing the struct into the front of the new bucket, i.e., relinking its left and right pointers. To increase the efficiency, if an ST struct remains in the same bucket after the update, only its time of occurrence will be changed. It is a simple practice to implement the HL algorithm in C/C++.

We remark that according to our test, it seems to be less efficient to implement buckets as linear arrays than to implement them as linked lists. Although linear
Arrays are more cache friendly than linked lists, we have to frequently move structs from one bucket to the other instead of just relinking pointers. In addition, to maintain the linear array data structure, when removing one element from the bucket, the last element have to be moved to fill the empty slot. As a result, we observed $\sim 10\%$ decrease of the performance when implementing buckets as linear arrays.

![Figure 1. TimeArray and BucketArray for $Q = 2$ and $\tau = 1$](image)

The HL algorithm can be summarized as follows:

**Step 1:** Initialization:
(a) Initialize $R_i$s
(b) Generate the dependency graph $G$.
(c) Generate time of occurrence $t_1, \ldots, t_N$.
(d) Initialize TimeArray, RateArray, and BucketArray

**Step 2:** Choose proper parameters $\tau$ and $Q$.

**Step 3:** Distribute time of occurrence to corresponding buckets BucketArray[0] to BucketArray[Q].

**Step 4:** For $i = 0$ to $Q - 1$
While BucketArray[i] is nonempty:
(a) Find the least time of occurrence $t_l$ within this bucket
(b) Make update according to $T_l$
(c) Follow the dependency graph $G$ to update all affected rate functions in RateArray
(d) Update time of occurrence of all affected events as
$$t_i^{\text{new}} = (t_i^{\text{old}} - t_l) \cdot \frac{R_i^{\text{old}}}{R_i^{\text{new}}} + t_l$$

and move affected elements in TimeArray to the corresponding buckets.
(e) Generate a new $t_l$ and move TimeArray[l] to the corresponding bucket

**Step 5:** Update the time intervals of each bucket and go to Step 3, or quit.

4.2. Analysis of complexity.

(1) **Average computational cost:** Assume $N$ is a large number and all $R_i$s are bounded from above by some constant independent of $N$, then for a suitable $\tau \sim O(1)$, $O(N)$ events occur in each step with time span $\tau$. The
computational cost of a bucket redistribution is $N$. If we choose $Q \sim O(N)$, then the number of events in each bucket can be (very) roughly approximated by a Poisson distribution with $O(1)$ mean. Assume the dependency graph is sparse such that the out degrees of all vertices are also $O(1)$. Then the average cost of updating each bucket has only computational cost $O(1)$. Therefore the total computational cost in one step is $O(N) + Q \times O(1) = O(N)$. This makes the computational cost of the HL algorithm be $O(1)$ per event.

If the dependency graph is not sparse in a way that the average out degree of vertices is $D$ which is possibly dependent of $N$, then the average cost of updating each bucket has computational cost $O(D)$. This brings the total computational cost in one step to $O(DN)$ and the computational cost per event to $O(D)$. We remark that the average computational cost per event of NRM is $O(D \log N)$ for Markov processes with non-sparse dependency graphs.

(2) **Worst case analysis:** In the worst case, which means all events are distributed into the same bucket, the HL algorithm is equivalent to the first reaction method (FRM). Therefore, the computational cost per event in the worst case is $O(N)$. However, we remark that this worst case appears with an extremely low probability. If we choose $Q \sim O(N)$, the probability that $O(N)$ events are placed into a single bucket is $\sim O(N^{-N})$.

### 4.3. Discussion of issues.

(1) **Choice of parameters:** In our simulations we choose $\tau$ such that about $N/5 \sim N/2$ clocks will ring in one step. The value of $Q$ is chosen in a way that each bucket contains $2 \sim 3$ events in average. These may not be optimal parameters. However in our simulations we find that the performance of the HL algorithm is not very sensitive with respect to small change of parameters. We will investigate efficient methods to choose optimal parameters of the HL algorithm in our subsequent works.

(2) **Possible improvements:** There are two places to further improve the HL algorithm. The first potential improvement is at the level of implementation. One can replace the linear search in each bucket by a binary search. It is easy to check that the expectation of the largest number of events stored in a bucket is $O(\log N)$. Therefore a binary search could potentially increase the performance of the algorithm. As mentioned before, it is slightly less efficient to implement buckets as linear arrays than to implement them as linked lists. However, we expect some improvement buckets are implemented as linear arrays, with binary heaps constructed on them. ( It is a standard knowledge that implementing a binary heaps as a linked list is significantly more complicated than implementing that as a linear array. ) We did not present this improvement in the present paper because it does not change the $O(1)$ per event computational cost of the HL algorithm. Plus due to the overhead of constructing the binary heap, we expect the improvement can only be seen for very large scale problems (i.e. very large $N$).
The second improvement is for Markov jump processes with sparse dependency graph. When the dependency graph is a large sparse graph, times of occurrence stored in each bucket will not affect each other in a very high probability ($\sim 1 - O(1/N)$). Therefore, in most situations we can update times of occurrence in each bucket sequentially from the head of the list without searching for the least time of occurrence. This improvement does not change the computational complexity of the algorithm either. In fact, because in average a bucket only contains $2 \sim 3$ events, we only observed a minor increase of simulation speed at the cost of much more complicated programming code. However, this idea can be used in the parallelization of our algorithm, as explained in (3).

(3) **Parallelization:**

Although the fact that events in one bucket are “almost independent” does not significantly improve the performance of the HL algorithm running on a single CPU, it can be used to parallelize this algorithm. In fact, the HL algorithm is very compatible with parallel programming if the dependency graph is sufficiently sparse. The idea is to make the number of buckets $Q \sim O(N/p)$, where $p$ is the number of CPUs, and divide one bucket into $p$ sub-buckets. During the bucket redistribution, we can evenly distribute events into sub-buckets. Suppose the dependency graph is sufficiently sparse, then events placed in each buckets will not affect each other with a relatively high probability. Therefore in most of the running time, different CPUs can update events in their own sub-buckets independently.

The principle of the parallelization is straightforward. However there are lots of details remain to be studied. The first issue is on implementation of the algorithm. With a low but strictly positive probability, an event in one bucket can affect events in the same bucket (but probably different sub-bucket), which will cause intensive communication between threads. Therefore it is crucial to design algorithms that can identify and deal with these small probability events with the lowest overall overhead. The second issue is the choice of parameters. As mentioned before, in the present paper we simply make each bucket contain $2 \sim 3$ events in average. However this shouldn’t be the case when running on parallel machines. In the future, we will write a separate paper to introduce the parallel HL algorithm. Optimization of parameters in various settings, including on parallelized platforms, will also be included in our consequent work.

5. **Numerical Examples**

5.1. **Introduction to models.** We choose the following three models to test the performance of the HL algorithm.

**A. Generalized KMP model** There are numerous stochastic processes in the field of statistical mechanics fit the general description of $X_t$ in Section 2, such as the simple inclusion process (SIP), simple exclusion process (SEP) and its variants [3, 11, 12], and many stochastic processes that models the microscopic heat conduction...
As the purpose of the present paper is to introduce the algorithm, we will choose the following generalized KMP model as a test case of our simulation. KMP model is a stochastic model proposed in [20] that models the microscopic energy transport, from which a microscopic derivation of Fourier’s law was carried out. The main feature of the generalized KMP model (See [19, 22] for details) here is its energy dependent clock rates, which makes simulations no longer trivial.

The KMP model models the energy transport in an 1d-chain of \( N \) oscillators coupled with two heat baths whose temperatures are \( T_L \) and \( T_R \), respectively. We only take note of energy carried by each oscillator, denoted by \( x_1, \ldots, x_N \), respectively. The Markov chain generated by the KMP model can be described as follows. An exponential clock with rate \( R(x_i, x_{i+1}) \), \( i = 0 \sim N \) is associated with each pair of adjacent oscillators (let \( x_0 = T_L \) and \( x_{N+1} = T_R \)). When the clock rings, the energy stored in corresponding pair of oscillators is pooled together, repartitioned randomly, and redistributed back to the oscillators. The energy redistribution satisfies

\[
(x_i', x_{i+1}') = (p(x_i + x_{i+1}), (1-p)(x_i + x_{i+1}))
\]

where \( p \) is a uniform random variable on \((0, 1)\) that is independent of everything else, and \( x_i' \) is the energy carried by oscillator \( i \) after the update. If clock 1 (resp. clock \( N \)) rings, the energy of the first (resp. last) oscillator exchanges energy with an exponential random variable with mean \( T_L \) (resp. \( T_R \)) in the same way.

We let \( T_L = 1.0, T_R = 2.0 \), and \( R(x_i, x_{i+1}) = \sqrt{x_i + x_{i+1}} \) and simulate the KMP model with varying \( N \)s.

It is worth to mention that Markov jump processes arise in other statistical mechanics models may not satisfy the generic description of \( X_t \) in Section 2. However, they can still be simulated by the HL algorithm efficiently. One example is the random halves model proposed in [13] and its modification [23], in which the number of clock varies randomly with time instead of being a constant. When the number of exponential clocks changes by one, the cost of maintaining bucket data structure is merely \( O(1) \). In contrast, in NRM, the computational cost of maintaining the binary heap after adding/removing one exponential clock is \( O(\log N) \).

B. Chemical reaction network

The second example we choose is a chemical reaction network. Stochastic chemical reaction networks is a very important class of Markov jump process. It is well known that many algorithms like DM, FRM, NRM, and CR are all originally developed for the stochastic simulation of chemical reactions. To compare the performance with other algorithms, we use the same example in [27] with minor modification.

Consider a large chemical reaction network with \( N \) reactions. For the sake of simplicity we only take note of rate of reactions, or the propensity. The dependency graph is randomly generated in a way that each reaction affects \( m \) other reactions, where \( m \) is a random integer number uniformly distributed between 1 and 30. After the occurrence of each reaction, all affected reactions are updated in a way that the propensity is replaced by a random number uniformly distributed on \([0, 2]\). The only difference between our example and the example presented in [27] is that we do not need to bound the ratio of the maximum rate and the minimum rate of reactions. Reaction rates in [27] are set to be between 1 and \( 1 \times 10^6 \) initially, and jump \( \pm 5\% \)
when “affected” by another reaction. We note that this example is different from
the usual real-world chemical reaction networks. It is used only for the purpose of
testing algorithms.

C. Reaction-diffusion system
The third example we choose is the Gray-Scott model, which is a reaction-diffusion system. Gray-Scott model is well known as its
patter formation phenomenon [28], as seen in Fig 2. The reaction part of the model
involves two species $U$ and $V$ with reactions

$$
\begin{align*}
U + 2V & \xrightarrow[K_1(\Omega)]{} 3V \\
U & \xrightarrow[K_f]{\emptyset} \\
V & \xrightarrow[K_f]{\emptyset} \\
\emptyset & \xrightarrow[K_f u_0(\Omega)]{} U
\end{align*}
$$

where $\Omega$ is a parameter that means the scale of the system, $k_1(\Omega) = \hat{k}_1 \Omega^{-2}$,
$u_0(\Omega) = \hat{u}_0 \Omega$, and $\hat{k}_1, K_f, K_2, \hat{u}_0$ are all parameters. Besides reactions, $U$ and $V$ has
diffusion rates denoted by constants $D_U$ and $D_V$, respectively.

In the simulation, we divide the domain into $K \times K$ subvolumes for varying $K$ and
take note of number of molecules of species $U$ and $V$ in each subvolume, denoted by
$U_{i,j}$ and $V_{i,j}$, respectively, where $i, j$ ranges from 1 to $K$. Therefore six exponential
clocks are associated to each subvolume with rates $U_{i,j}V_{i,j}^2K_1(\Omega)$, $U_{i,j}K_f$, $V_{i,j}K_f$,
$K_f u_0(\Omega)$, $U_{i,j}D_U$, and $V_{i,j}D_V$, respectively. The size of this system is $N = 6K^2$. The
values of parameters are taken as $\Omega = 250$, $K_f = 0.0055$, $K_2 = 0.0205$, $D_V = 0.002$,
$D_U = 0.001$, and $\hat{k}_1 = \hat{u}_0 = 1$.

![Figure 2](image)

**Figure 2.** Pattern formation of the Gary-Scott model. Contour plot
of $U$ species at $T = 1500.0$. Left: $K = 100$; Right: $K = 200

5.2. Performance of the HL Algorithm.
A. Speed of simulation

The first batch of numerical simulations concern the speed of algorithm. We implement three models introduced in the previous subsection with both the HL algorithm and the benchmark NRM algorithm. Both algorithms are implemented in C, with C++ I/O for the sake of simplicity of coding. All performance tests are run on a 2012 Macbook Pro laptop with an Intel Core I7-3615QM CPU and 8 GB memory.

Instead of stopping after simulating a certain number of events, we choose to simulate all examples up to \( T = 10 \), as simulating Markov processes up to different times may bias the result. Performances of the two algorithms are measured in seconds per million events and compared, with system size \( N \) ranging from 100 to \( 1 \times 10^6 \sim 1 \times 10^7 \). The choice of parameters of the algorithm are \( \tau = 0.2, Q = N/10 \) for the generalized KMP model, \( \tau = 0.1, Q = N/20 \) for the chemical reaction network, and \( \tau = 0.5, Q = N/2 \) for the reaction-diffusion system. These may not be the optimal parameters. But from our test, the speed of simulation is not sensitive with respect to the small change of parameters.

We plot seconds of simulating per million events vs. \( N \) in linear-log plots. As shown in Figure 3, for large \( N \), the speed of the HL algorithm is 3 to 4 times faster than that of the NRM algorithm. We remark that although having the same computational complexity, the HL algorithm is slightly faster than the CR algorithm. For example, for a chemical reaction network with \( 1 \times 10^6 \) reactions, the CR algorithm is \( \sim 2 \) times faster than the NRM algorithm \( [27] \), while our HL algorithm is \( \sim 3 \) times faster than the benchmark NRM algorithm.

B. Statistics of computer operations
We would like to mention that the speed of algorithms in terms of seconds per million events depends on many things beyond the efficiency of algorithms. Details of implementation, the compiler, the operating system, and the size of CPU cache...
can significantly change the speed of algorithm. Therefore it is important to study the number of operations per event for the HL algorithm.

We collected data of computer operations of all three numerical examples. In terms of generating new random numbers and updating time of occurrence of affected clocks, the HL algorithm has no difference from other mainstream algorithms. Therefore, we only focus on the number of comparison operations when linearly searching a bucket and the number of moves of events between buckets. In all three cases, numerical simulations confirm our observation that the HL algorithm has constant computational cost per event. The total number of comparisons plus moves per event are $\approx 5.0$ for the generalized KMP model, $\approx 10.5$ for the reaction diffusion system, and $\approx 27.5$ for the chemical reaction network, respectively. In addition, we demonstrate Figure 6 for the generalized KMP model as an example of $O(1)$ algorithm vs. $O(\log N)$ algorithm (the NRM algorithm), in which the number of moves of events are further broken down into moves with relinking pointers and moves without relinking pointers (i.e. within the same bucket). The numbers of operation of NRM shown in the figure are the numbers of swaps in the binary heap.

![Figure 6. Number of operations per events in NRM and HL](image)

6. Conclusion

In the present paper, we introduced a fast simulation algorithm, namely the HL algorithm, for a class of Markov jump processes arise in many scientific fields. As a Markov jump process proceeds as a sequence heterogeneous, state-dependent exponential random times, the main issue in simulating the Markov jump process is to
identify the next time of occurrence among many exponential random times. The main idea of the HL algorithm is to bucket sort all times of occurrence covered by a certain time step with length $\tau$, update all buckets sequentially, and leap forward by $\tau$. For Markov chains with sparse dependency graph, the average computational cost per event of the HL algorithm is $O(1)$, independent of the number of exponential clocks. If on the other hand, the dependency graph is non-sparse, then the average computational cost per event becomes $O(D)$, where $D$ is the average out degree of the dependency graph.

For a large class of models, the HL algorithm has certain advantages over existing stochastic simulation algorithms. It has low computational complexity. In addition, it is easy to implement, and very compatible with parallel machines. We test the performance of the HL algorithm on three different large-scale Markov jump processes: the generalized KMP model, the chemical reaction network, and the stochastic reaction-diffusion system. Our simulation results show significant advantage of the HL algorithm over the benchmark algorithm (NRM). In addition, we verified numerically that the average computer operations per event is constant for the HL algorithm.

The main purpose of this paper is to introduce the HL algorithm. We remark that the performance of the HL algorithm can still be potentially improved in several aspects, as introduced in Section 4.3. It is also useful to trim the HL algorithm for specific problems, such as multiple time scale systems and systems with varying number of exponential clocks. Those issues will be studied in our subsequent works. In addition, we will investigate some subtle issues like optimal choices of parameters in our future works.

Acknowledgement

The authors would like to thank Lai-Sang Young and Kevin Lin for many enlightening discussions and valuable suggestions.

References

[1] Farid F Abraham and George M White, *Computer simulation of vapor deposition on two-dimensional lattices*, Journal of Applied Physics 41 (1970), no. 4, 1841–1849.
[2] JR Beeler Jr, *Displacement spikes in cubic metals. i. $\alpha$-iron, copper, and tungsten*, Physical Review 150 (1966), no. 2, 470.
[3] C Boldrighini, G Cosimi, S Frigio, and M Grasso Nunes, *Computer simulation of shock waves in the completely asymmetric simple exclusion process*, Journal of Statistical Physics 55 (1989), no. 3-4, 611–623.
[4] Federico Bonetto, Joel L Lebowitz, and Luc Rey-Bellet, *Fourier’s law: A challenge for theorists*, arXiv preprint math-ph/0002052 (2000).
[5] Alfred B Bortz, Malvin H Kalos, and Joel L Lebowitz, *A new algorithm for monte carlo simulation of ising spin systems*, Journal of Computational Physics 17 (1975), no. 1, 10–18.
[6] Yang Cao, Dan Gillespie, and Linda Petzold, *Multiscale stochastic simulation algorithm with stochastic partial equilibrium assumption for chemically reacting systems*, Journal of Computational Physics 206 (2005), no. 2, 395–411.
[7] Yang Cao, Daniel T Gillespie, and Linda R Petzold, *Efficient step size selection for the tau-leaping simulation method*, The Journal of chemical physics 124 (2006), no. 4, 044109.
[8] Yang Cao, Hong Li, and Linda Petzold, *Efficient formulation of the stochastic simulation algorithm for chemically reacting systems*, The journal of chemical physics 121 (2004), no. 9, 4059–4067.

[9] Yang Cao and Linda Petzold, *Slow-scale tau-leaping method*, Computer methods in applied mechanics and engineering 197 (2008), no. 43, 3472–3479.

[10] Vincent Danos, Jérôme Feret, Walter Fontana, and Jean Krivine, *Scalable simulation of cellular signaling networks*, Programming Languages and Systems, Springer, 2007, pp. 139–157.

[11] Bernard Derrida, *An exactly soluble non-equilibrium system: the asymmetric simple exclusion process*, Physics Reports 301 (1998), no. 1, 65–83.

[12] Bernard Derrida, Steven A Janowsky, Joel L Lebowitz, and Eugene R Speer, *Exact solution of the totally asymmetric simple exclusion process: shock profiles*, Journal of statistical physics 73 (1993), no. 5-6, 813–842.

[13] J-P Eckmann and L-S Young, *Nonequilibrium energy profiles for a class of 1-d models*, Communications in Mathematical Physics 262 (2006), no. 1, 237–267.

[14] Johan Elf and Måns Ehrenberg, *Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases*, Systems biology 1 (2004), no. 2, 230–236.

[15] Michael A Gibson and Jehoshua Bruck, *Efficient exact stochastic simulation of chemical systems with many species and many channels*, The journal of chemical physics A 104 (2000), no. 9, 1876–1889.

[16] Daniel T Gillespie, *A general method for numerically simulating the stochastic time evolution of coupled chemical reactions*, Journal of computational chemistry 22 (1976), no. 4, 403–434.

[17] **, *Exact stochastic simulation of coupled chemical reactions*, The journal of physical chemistry 81 (1977), no. 25, 2340–2361.

[18] **, *Approximate accelerated stochastic simulation of chemically reacting systems*, The Journal of Chemical Physics 115 (2001), no. 4, 1716–1733.

[19] Alexander Grigo, Konstantin Khanin, and Domokos Szasz, *Mixing rates of particle systems with energy exchange*, Nonlinearity 25 (2012), no. 8, 2349.

[20] C Kipnis, C Marchioro, and E Presutti, *Heat flow in an exactly solvable model*, Journal of Statistical Physics 27 (1982), no. 1, 65–74.

[21] M Kotrla, *Numerical simulations in the theory of crystal growth*, Computer Physics Communications 97 (1996), no. 1, 82–100.

[22] Yao Li and Lai-Sang Young, *Existence of nonequilibrium steady state for a simple model of heat conduction*, Journal of Statistical Physics 152 (2013), no. 6, 1170–1193.

[23] **, *Nonequilibrium steady states for a class of particle systems*, Nonlinearity 27 (2014), no. 3, 607.

[24] Carl Adam Petri, *Kommunikation mit automaten*, (1962).

[25] Rajesh Ramaswamy, Nélido González-Segredo, and Ivo F Sbalzarini, *A new class of highly efficient exact stochastic simulation algorithms for chemical reaction networks*, The Journal of chemical physics 130 (2009), no. 24, 244104.

[26] Rajesh Ramaswamy and Ivo F Sbalzarini, *A partial-propensity variant of the composition-rejection stochastic simulation algorithm for chemical reaction networks*, The Journal of chemical physics 132 (2010), no. 4, 044102.

[27] Alexander Slepoy, Aidan P Thompson, and Steven J Plimpton, *A constant-time kinetic monte carlo algorithm for simulation of large biochemical reaction networks*, The journal of chemical physics 128 (2008), no. 20, 205101.

[28] Matthias Vigiedus and Bernd Meyer, *Stochastic simulations of pattern formation in excitable media*, PloS one 7 (2012), no. 8, e42508.

[29] Arthur F Voter, *Introduction to the kinetic monte carlo method*, Radiation Effects in Solids, Springer, 2007, pp. 1–23.

[30] E Weinan, Di Liu, and Eric Vanden-Eijnden, *Nested stochastic simulation algorithm for chemical kinetic systems with disparate rates*, The Journal of chemical physics 123 (2005), no. 19, 194107.
[31] Jin Yang, Michael I Monine, James R Faeder, and William S Hlavacek, *Kinetic monte carlo method for rule-based modeling of biochemical networks*, Physical Review E 78 (2008), no. 3, 031910.

YAO LI: COURANT INSTITUTE OF MATHEMATICAL SCIENCES, NEW YORK UNIVERSITY, NEW YORK, NY 10012, USA  
*E-mail address: yaoli@cims.nyu.edu*

LILI HU: THESYS TECHNOLOGY LLC, NEW YORK, NY 10003, USA  
*E-mail address: lilyhu86@gmail.com*