Parallel Kinetic Monte Carlo

Anya Katsevich
Terrence Alsup

1. Introduction to Kinetic Monte Carlo

Kinetic Monte Carlo (KMC) is a simulation method that can be used to model the evolution of the surface of a crystal. Consider a 1-D lattice $\{1, \dots, L\}$ of size L. At each site ℓ on the lattice we have h_ℓ atoms stacked on top of each other. The heights h_ℓ of these stacks determine the surface of the crystal. For a given state $\vec{h} = (h_1, \dots, h_L)$ of the lattice there is an associated energy and atoms move across the surface of the crystal based on how this energy will change (see Figure 1). Atoms are more likely to jump more frequently in regions where there are large height differences between neighboring stacks. The serial KMC algorithm works by simulating a Markov jump process $\vec{h}(t) = (h_1(t), \dots, h_L(t))$ on the space of all possible lattice states and is summarized in the following three steps.

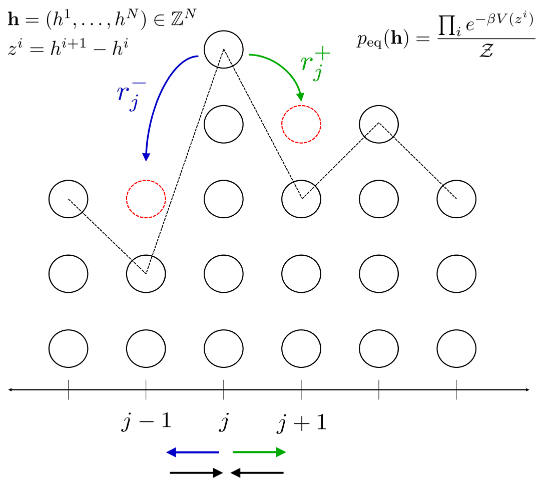
$\mathbf{h} = (h^1, \dots, h^N) \in \mathbb{Z}^N$
$z^i = h^{i+1} - h^i$
$p_{\text{eq}}(\mathbf{h}) = \frac{\prod_i e^{-\beta V(z^i)}}{\mathcal{Z}}$
r_j^-
r_j^+
$j-1$
j
$j+1$

Figure 1: Surface diffusion on 1D lattice; an event occurs at site j when the topmost atom jumps left or right, at a rate of $r_j^\pm$.

Serial KMC Input: Initial lattice state $\vec{h}$ and time $t = 0$
While $t < T$ do

1. For the given state $\vec{h}$, compute the current values of all rates, r_i, of events E_i. The total rate of all events is $R = \sum_i r_i$.
2. Draw a waiting time $\Delta t \sim \text{Exp}(R)$ for the next event to occur, and update the time $t \mapsto t + \Delta t$.
3. Draw an event E from the discrete distribution $\mathbb{P}(E_i) = \frac{r_i}{R}, i = 1, 2, \dots$. Update the state $\vec{h}(t) \mapsto \vec{h}(t + \Delta t)$, where $\vec{h}(t+\Delta t)$ denotes the state of the lattice after the event E has occurred.

The rates for jumping left or right from lattice site i are equal and computed by the formula

$$r_i = \frac{1}{2} \exp\left(K\left(h_{i+1} - 2h_i + h_{i-1} + 1\right)\right), \tag{1}$$

where K is proportional to the inverse temperature of the system. Note that the jump rates depend on a discrete Laplacian, so the system will favor states with uniform heights.

In practice we are interested in how the crystal surface evolves when L is very large. In the limit as $L \to \infty$ we can re-scale the height, domain, and time as $h(x,t) = \frac{1}{L} h_{\lfloor Lx \rfloor}(L^4 t)$, and the continuous height profile $h(x,t)$ will satisfy the following PDE describing the evolution of h:

$$\partial_t h(t,x) = -\frac{K}{2} \partial_{xxx} \left[\sigma_F \left(\partial_x h(t,x) \right) \right], \tag{2}$$

where σ_F is the free energy of the surface (see [1] for more details). However, taking L to be very large is both computationally expensive and wasteful. The serial KMC algorithm requires that only one atom jumps at a time and then recomputes all of the jump rates to draw a new event. This means that even atoms very far away from each other must wait to move, even though atoms can only jump to their immediate neighbors. Taking advantage of this fact will allow us to parallelize the jump process and achieve speedup.

2. Parallel Algorithm

Parallel KMC solves this problem by carrying out events in different sections of the lattice grid simultaneously. This allows us to simulate the same length of physical time in a shorter amount of computer time. In parallel KMC, the lattice is split into subregions, or blocks, which are assigned to different processors. Each processor has two ghost sites which keep track of the height of the crystal at the site immediately to the left of its leftmost site and to the right of its rightmost site. The processors run a modified version of serial KMC in parallel, then exchange boundary height values to update their ghost sites, as well as other information (explained below). Then, the processors proceed with the next iteration of serial KMC. The modification to serial KMC in each block is necessary to ensure that two processors do not simultaneously update the crystal state at their shared boundary. This is done by dividing each block into a left and right section. Each processor first carries out events in its left section for a designated amount of time t_{stop}, so that particles in two different blocks could not possibly jump to the same boundary. Then, boundary information is communicated, after which each processor carries out events in its right section. Below is a schematic of this approach for a two-dimensional implementation of parallel KMC.

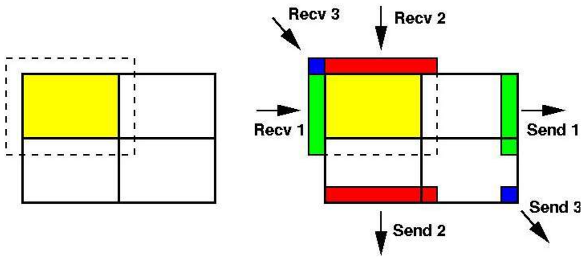

Figure 2: A processor works on the yellow section of its block, then sends and receives boundary information from other processors before moving on to work on the next section

The amount of physical time t_{stop} that each processor runs independently before exchanging information with other processors is determined by the maximum rate sum $R^0_{\max}$ in the left (resp. $R^1_{\max}$ in the right) section of each block. If the rate sum is high in a section of a block, then events will occur at a faster rate there. Using $R_{\max}$ to determine t_{stop} sets a (probabilistic) limit on the number of events that can occur before information is exchanged. However, it also means that there must be an all-reduce type communication among processors after every parallel run of serial KMC. On a high level, one cycle of parallel KMC on processor k proceeds as follows:

Parallel KMC [2],[3]

Input: current physical time t, current values of $R^{0,1}_{\text{loc}}$ (local sum of rates in each section), $R^{0,1}_{\max}$, t_{stop}, and current crystal state $\vec{h}_k(t)$ (the block of h known to processor k including ghost sites and interior points)

1. Run **Serial KMC** until time $t = t + t_{\text{stop}}$ in the left section of the block and update $R^{0,1}_{\text{loc}}$.
2. Send boundary information at the leftmost points to processor $k-1$ and receive boundary information from the leftmost points of processor $k+1$. Recompute boundary rate values using this information.
3. Compute the new value of $R^1_{\max}$ and $t_{\text{stop}} = \frac{C}{R^1_{\max}}$ (C is a global fixed constant).
4. Run **Serial KMC** until time $t + t_{\text{stop}}$ in the right section of the block and update $R^{0,1}_{\text{loc}}$.
5. Send boundary information at the rightmost points to processor $k+1$ and receive boundary information from the rightmost points of processor $k-1$. Recompute boundary rate values using this information.
6. Compute the new value of $R^0_{\max}$ and $t_{\text{stop}} = \frac{C}{R^0_{\max}}$

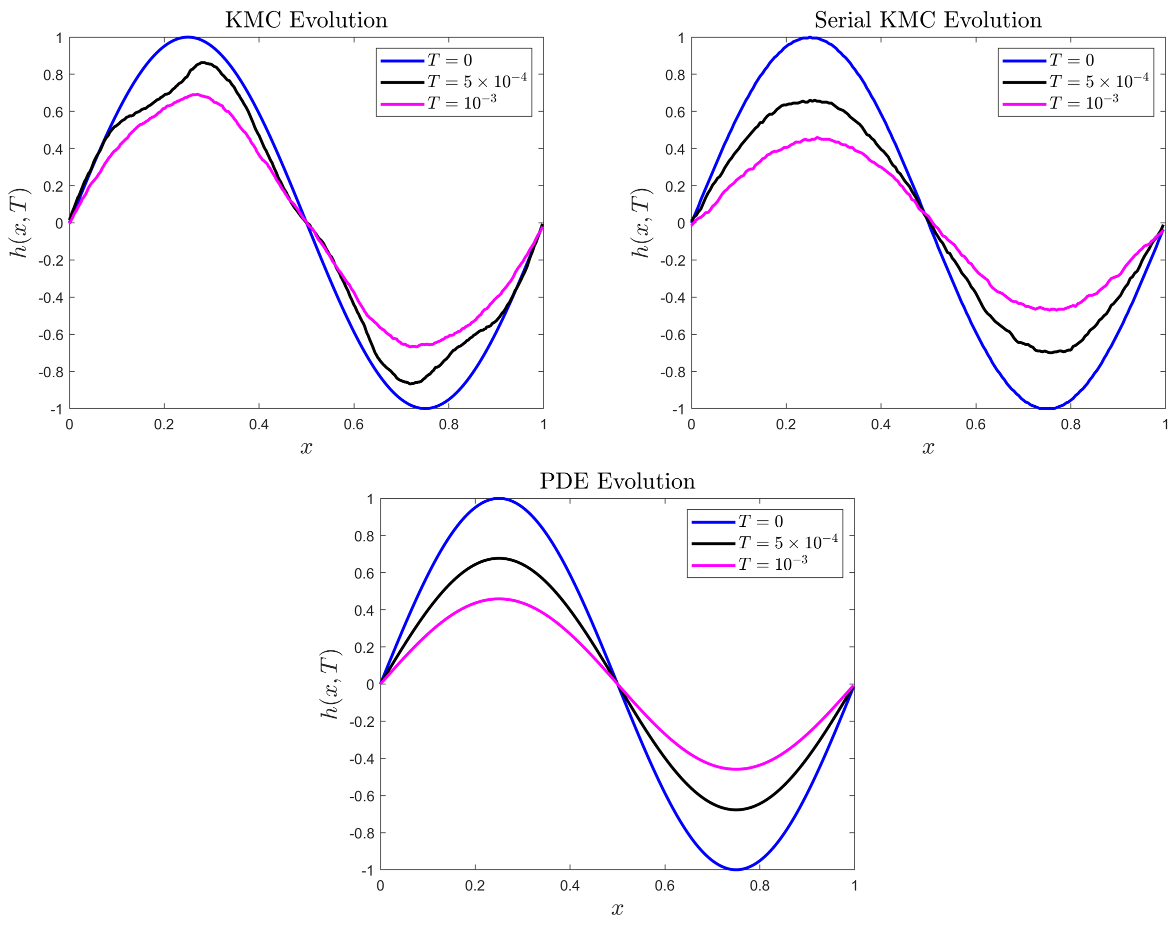

Figure 3: Evolution of crystal surface simulated with parallel KMC simulation compared to serial KMC simulation and evolution of PDE solution. Here the initial condition was $h_0(x) = \sin(2\pi x)$.

3. Performance of Parallel KMC

We implemented the parallel algorithm using MPI, and ran the code on the Prince and Courant (`crunchy1` and `crunchy3`) computing clusters. The run-time plots were generated based on runs on the Prince cluster.

Note that the parallel algorithm only approximates the true stochastic process, since the time for an event to occur at any point in the lattice should depend on the sum of the transition rates in the whole lattice. If we execute events in parallel without communicating the rates in the whole grid at every step, the algorithm is necessarily unexact. However, physical principles suggest that far away particles on the lattice should not influence each other, so the algorithm should result in a reasonable approximation for large crystal sizes. Figures 3 and 4 show that KMC approximates the PDE reasonably well for a crystal size $L = 400$. As to be expected, the serial KMC algorithm results in a process that is closer to the PDE evolution. For smaller L, the parallel algorithm results in a process that behaves totally unpredictably.

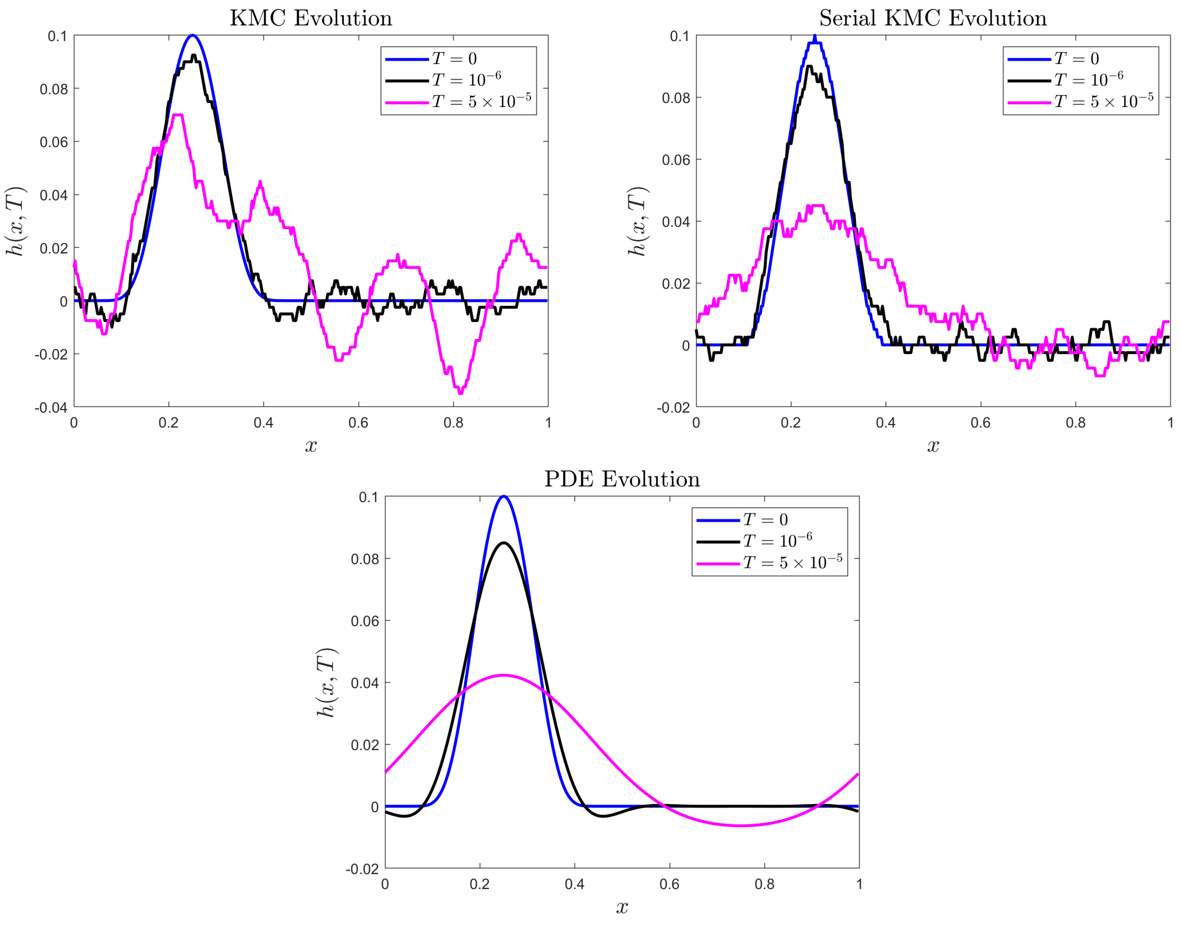

Figure 4: Evolution of crystal surface simulated with parallel KMC simulation compared to serial KMC simulation and evolution of PDE solution for an initial condition with compact support.

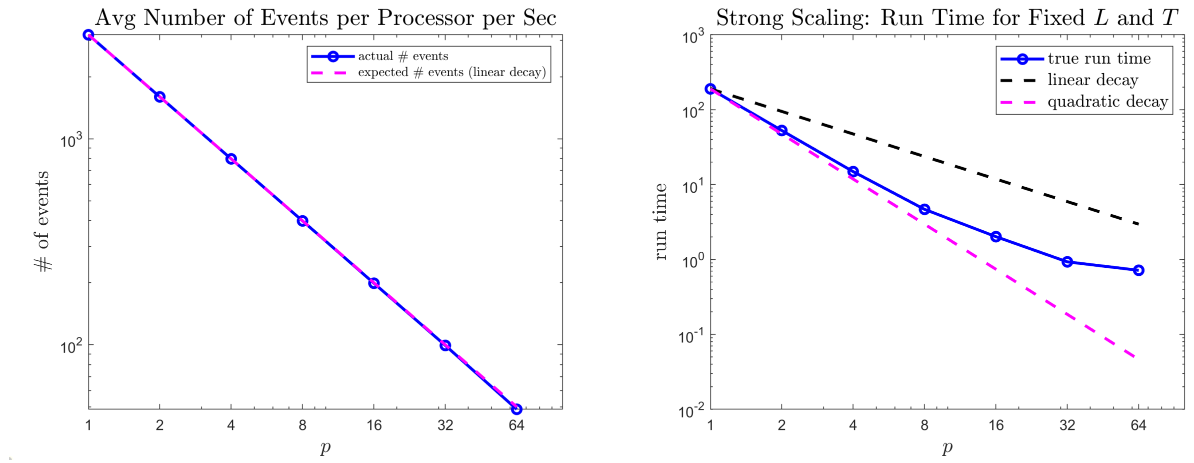

Figure 5: Strong scaling algorithm performance (left), and number of events per processor per second (right). and. Note that the run time decreases faster than linearly as the number of processors increases. This is because both the number of events and the computational effort per event decreases linearly as the number of processors increases.

	Complexity
Microscopic time	L^4T
L_{loc}	L/P
R_{loc}, $R_{\max}$	$O(L_{\text{loc}})$
t_{stop}	$cL/R_{\max}$
# Events of serial KMC / processor	$O(L^4T \times R_{\text{loc}}) = O(L^5/P)$
Computational effort / event	$O(L_{\text{loc}}) = O(L/P)$
Total computational effort / processor	$O(L^4T \times R_{\text{loc}} \times L_{\text{loc}}) = O(L^6/P^2)$
# Communications / KMC section update	$O(P)$
# Updates	$O(L^4T/t_{\text{stop}}) = O(L^4/P)$
Total # communications	$O(L^4)$

Table 1

The table above shows the scaling of expected computational complexity and expected number of communications with L and P. The only memory access-heavy part of the code is in sampling from the probability distribution of the rates when selecting an event. However, since this requires accessing each element in the array of rates sequentially, this should not be too expensive.

3.1. Strong Scaling

For the strong scaling experiment we took a fixed value of T and L. Increasing the number of processors by a factor of P decreases the complexity of a single event update by a factor of P. This is because a single event requires sampling from a distribution with L_{loc} rates (one per lattice site), which has complexity $O(L_{\text{loc}})$.

It also decreases the number of events per processor by a factor of P, since we want the total number of events in time T to stay roughly constant. Thus, we should ideally expect a speedup of P^2. Figure 5 shows that the actual performance is reasonably close to this ideal speedup.

3.2. Weak Scaling

The problem has inherently poor weak scaling, for the following reason: microscopic time scales like L^4 with macroscopic time T, and the dynamics of interest are macroscopic, i.e. one is interested in taking L large and T fixed (rather than decreasing with L). If we were to take T fixed in a weak scaling experiment, (in which L_{loc} was fixed), increasing the number of processors by a factor of P would increase the microscopic time by a factor of P^4, but decrease the computational complexity by a factor of P^2, resulting in a net slowdown of P^2.

4. Conclusion

Parallel KMC can take advantage of the fact that the atoms only move locally, which for large crystals $L \gtrapprox 400$ gives a reasonable approximation to the true stochastic process. Since the strong scaling is so significant (a factor of P^2), we learned how effective it can be to run code on a high performance computing cluster like Prince. For example, running on 64 nodes can reduce the computation time from several hours to less than a minute. Surprisingly, implementing the communication between processors was much more straightforward than implementing the modified serial KMC portion of the code.

References

[1] J. Marzuola and J. Weare. *The relaxation of a family of broken bond crystal surface models.* Phys Rev E 88 (2013), 032403.

[2] S. Plimpton, C. Battaile, et.al. *Crossing the Mesoscale No-Man's Land via Parallel Kinetic Monte Carlo*, SAND20096226 Unlimited Release, October 2009

[3] I. Martin-Bragado, J. Abujas, P.L. Galindo, J. Pizarro *Synchronous parallel Kinetic Monte Carlo: Implementation and results for object and lattice approaches*, Nuclear Instruments and Methods in Physics Research B, 352 (2015) 2730