Parallelized Simulations of Grain-Surface Chemistry

William Burke, Drake Deuel, Esmail Fadae, Jamila Pegues

May 2019

1 Motivation

1.1 Background

The chemical composition and atmospheres of exoplanets are the target of new and upcoming telescope missions (e.g., NASA's TESS Mission, Ricker et al., 2014, launched in April 2018). This is in large part because exoplanets offer the opportunity to find other worlds that are habitable to, and may even host, Earth-like life. However, in order to understand the chemical composition of these exoplanets, and to be able to distinguish any biosignatures from abiotic false positives (Ballard, 2019), we need to understand how these exoplanets formed their chemical compositions in the first place. This understanding requires studying chemistry on much, much smaller scales: on the surfaces of dust grains. Asteroids, planetesimals, and planets form from the collision and accumulation of microscopic dust grains; therefore grain-surface chemistry, through which complex molecules can form and even build up in layers on a grain's surface, plays a crucial role in the chemical history of these planetary bodies.

Unfortunately, grain-surface chemistry is extremely difficult to study. Laboratory experiments would be the ideal method for exploring grain-surface chemistry; however, even just experimentally approximating the cold, dark, sparse conditions of space requires monetarily expensive laboratory setup and equipment. Computational simulations of grain-surface chemistry offer the potential to be extremely powerful tools for robustly exploring grain-surface chemistry in different astronomical environments. In practice, however, these simulations are also expensive, for several reasons: (1) chemistry happens quite quickly relative to grain growth, meaning that the simulation timesteps must remain short even as the timespan of the overall simulation grows; (2) atoms are small ($\sim 10^{-10}$m) relative to the radii of typical dust grains ($\sim 10^{-7}$m), such that the elements within the simulation must be small compared to the physical area represented by the simulation; and (3) in suitable physical conditions, atoms/molecules can accumulate on the grain surface over time, and the simulation must devote resources to keeping track of all of these particles at each timestep as the simulation progresses. These factors combine such that simulations of grain-surface chemistry quickly become computationally and temporally expensive as the size of the grain and the length of the simulation increase. To perform these simulations within reasonable time frames and resource budgets, we need an approach that minimizes the expenses without sacrificing simulation accuracy.

The work of Chang et al. (2005) shows that the major processes of grain-surface chemistry can be described in a Poisson framework. This means that grain-surface simulations can treat the movement of particles as independent events. Chang et al. (2005)

Parameter	Value	Reason for Value
$E_{bind,a}$	5.14982×10^{-21}J	Binding energy of H to the grain surface
$E_{surf,a}$	3.957×10^{-21}J	Diffusion barrier of H on the grain surface
m_a	$1.6738234 \times 10^{-27}$kg	Mass of H
n_a	10^6m^{-3}	Typical H number density of diffuse ISM
N_s	$2 \times 10^{18} \text{m}^{-2}$	Surface density of sites on the grain surface
r_{gr}	$0.5\sqrt{\frac{S}{\pi N_s}}$	Radius of the grain; scales with grain size
s_a	1	Sticking efficiency
S	*Variable*	Total number of sites on the grain
T_{gas}	*Variable*	Temperature of the gas
T_{grain}	*Variable*	Temperature of the grain

Table 1: Simulation parameters and values. The grain surface is assumed to be homogeneous olivine; all grain-specific values ($E_{bind,a}$, $E_{surf,a}$, N_s, and r_{gr}) are based on the values used for this surface type by Chang et al. (2005). The environment is assumed to be the diffuse interstellar medium (abbreviated as ISM).

used this framework to construct a continuous-time, random-walk simulation that explored the formation of molecular hydrogen (H_2) from hydrogen atoms (H) on a grain surface under different physical conditions. Though their publication was released over a decade ago, the approach of Chang et al. (2005) is still used in studies today; however, the scales of these studies have remained small in order to avoid computational and temporal expenses at larger problem sizes (e.g., Chang and Herbst, 2014; Cuppen et al., 2017; Wakelam et al., 2017).

In this work, we present a new grain-surface simulation code that simulates the formation of H_2 on grain surfaces. We take advantage of the Poisson framework of Chang et al. (2005) to optimize and parallelize the simulation. The code achieves a speed-up of !!!x, and its performance !!!x better than the serial case. The code can be expanded to handle more complex atoms, reactions, and grain-surface scenarios with little or no increase in computational and temporal cost.

2 Methodology

2.1 Grain-Surface Processes

Within typical grain-surface chemistry models, there are four major processes that a particle (i.e., an atom or molecule) can undertake (e.g., Cuppen et al., 2017). We include these processes within our simulation and briefly summarize them below. All processes are illustrated in Figure 1. We use the general equations as presented in the recent grain-surface chemistry review by Cuppen et al. (2017), but we focus on hydrogen and simulate a similar environment to that of Chang et al. (2005) to allow easy and consistent comparison between our results and their original work. Note that the purpose of this work was not to produce any new or accurate scientific results, but instead to develop an efficient baseline grain-surface simulation code, which can later be expanded to handle complex scientific grain-surface scenarios at drastically reduced computational and temporal costs.

All parameter values used for our simulation are listed in Table 1.

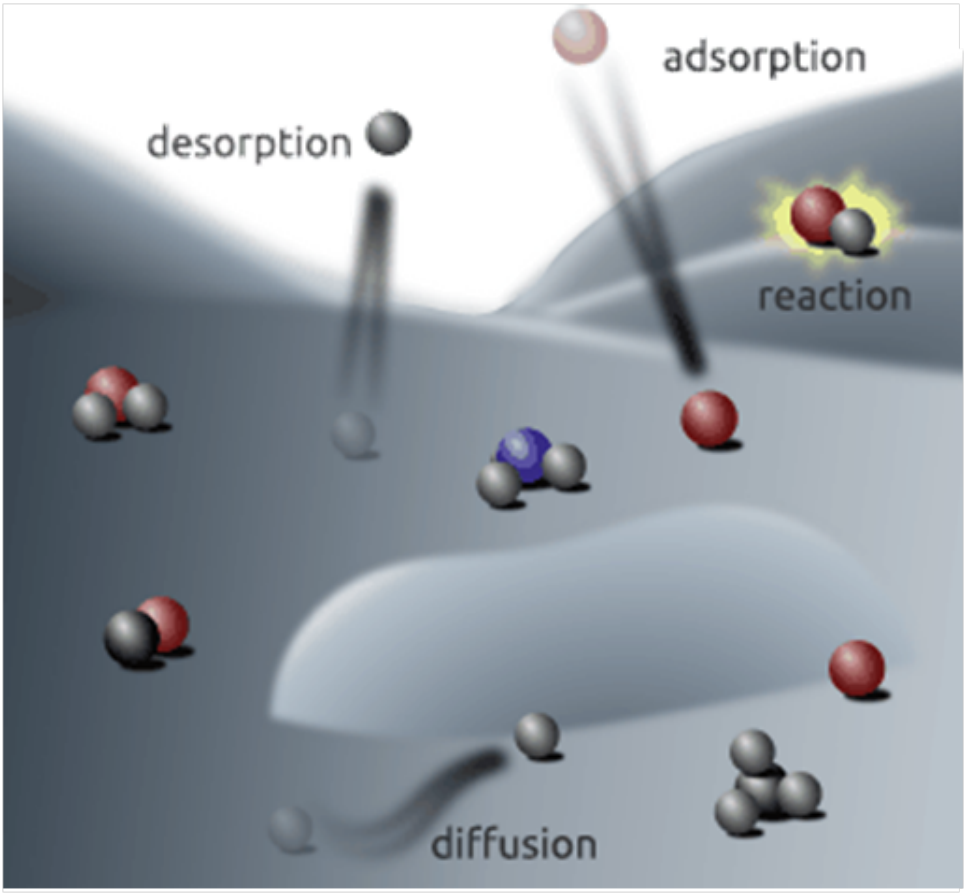


Figure 1: An illustration of the four major grain-surface processes included in our simulation. Reproduced from Figure 1 of Cuppen et al. (2017).

2.1.1 Adsorption

This is the process of a particle joining to a grain surface. The rate of this process (in units of [time^{-1}]) is as follows:

$$R_{ads} = s_a v_a n_a \pi r_{gr}^2, \tag{1}$$

where s_a is a sticking efficiency $\in [0, 1]$ that describes how likely the particle a will stick to the grain; n_a is the number density of a in the surrounding gas; r_{gr} is the radius of the grain; and v_a is the average thermal velocity of a in the gas:

$$v_a = \sqrt{\frac{8kT_{gas}}{\pi m_a}}, \tag{2}$$

where T_{gas} is the temperature of the gas, k is the Boltzmann constant, and m_a is the mass of the species a. T_{gas} and n_a are naturally environment-dependent (e.g., the interstellar medium will have very different temperatures and densities compared to a dense molecular cloud core).

2.1.2 Desorption (Thermal)

Thermal desorption is the process of a particle on a grain leaving the grain's surface. The rate of this process is given as:

$$R_{des} = \nu \exp\left(\frac{-E_{bind,a}}{kT_{gr}}\right), \tag{3}$$

where $E_{bind,a}$ is the binding energy of a species a to the grain surface; k is again the Boltzmann constant; T_{gr} is the temperature of the grain; and ν is the characteristic attempt frequency, given as:

$$\nu = \sqrt{\frac{2N_s E_{bind,a}}{\pi^2 m_a}}, \tag{4}$$

where N_s is the surface density of sites, or cells, on the grain surface; and m_a is the mass of a.

2.1.3 Diffusion

Diffusion is the process of a particle moving from one site on a grain to an adjacent site on a grain. Similarly to desorption, the rate of this process is given as:

$$R_{diff} = \nu \exp\left(\frac{-E_{surf,a}}{kT_{gr}}\right), \tag{5}$$

where $E_{surf,a}$ is the diffusion barrier, which depends on the surface of the grain itself. Similarly, the characteristic attempt frequency is

$$\nu = \sqrt{\frac{2N_s E_{surf,a}}{\pi^2 m_a}}, \tag{6}$$

2.1.4 Reaction

Reactions occur when two particles are in the same cell on the surface. There is a particle-dependent probability that indicates how likely the product of a reaction (such as H_2 from H + H) will then desorb from the surface after the reaction. This is a parameter in our model. For the results below, we assumed that the probability of H_2 desorbtion was 1.

2.2 Poisson Framework

Chang et al. (2005) describe grain-surface chemistry within a Poisson framework, which treats the adsorption, diffusion, and desorption of particles as Poisson events, which are completely independent of each other. Generically, the amount of time t that passes between two Poisson events follows an exponential curve, which Chang et al. (2005) calls a 'waiting-time distribution' (WTD):

$$\psi_a(t) = R\exp(-Rt), \tag{7}$$

where R is the rate of a process (e.g., desorption), and therefore R^{-1} is the average time that passes between the process described by R. The time t between two events of this process can then randomly be sampled as:

$$t = -\frac{\ln X}{R}, \tag{8}$$

where X is a random number sampled from a uniform distribution in (0, 1).

This framework could be used to calculate the time between events on the grain within a grain-surface simulation. However, there is an issue with the simplifying assumption made by Chang et al. (2005). Plugging in the equation for t, we find

$$\psi_a(t) = R\exp(-Rt) = R\exp(R\frac{\ln X}{R}) = RX, \tag{9}$$

This amounts to sampling the uniform distribution when calculating the inter-event times, which does not preserve the properties of a Poisson process that 1) the number of events in disjoint intervals are independent and 2) the number of events in a fixed interval t follows a Poisson distribution Pois(Rt).

Using Chang et al. (2005) as inspiration, we applied the Poisson framework in a way that more easily lends itself to parallelization and does not require any simplifying assumptions. For a given process with rate λ, the probability that the number events k=0 in a timestep t is as follows from Blitzstein and Hwang (2015)

$$P(k=0) = \frac{e^{-\lambda t}(\lambda t)^k}{k!} = e^{-\lambda t}, \tag{10}$$

This gives the probability of at least one event

$$P(k>0) = 1 - e^{-\lambda t}, \tag{11}$$

This applies directly to adsorbtion, but particles on a grain surface may undergo of desorbtion and diffusion. The probability of at least one event happening is then a function of the combined rate

$$P(k>0) = 1 - e^{-(\lambda_{desorbtion}+\lambda_{diffusion})t}, \tag{12}$$

Conditioning on an event happening, Blitzstein and Hwang (2015) tell us that the probability of is proportional to the rate

$$P(desorbtion|event) = \frac{\lambda_{desorbtion}}{\lambda_{desorbtion} + \lambda_{diffusion}}, \tag{13}$$

$$P(diffusion|event) = \frac{\lambda_{diffusion}}{\lambda_{desorbtion} + \lambda_{diffusion}}, \tag{14}$$

Thus, to accurately model the complex processes occuring on a grain surface, we need only choose a timestep such that the probability for of two events for a given poisson process is exceedingly low, or the probability of no events is very high. Since the probabilities are a nonlinear function of temperature and interaction energy, we enter the desired probability as a parameter to our model, in this case 99.99%. From this parameter, we calculate the timestep and then precompute the event probabilities, which are held constant for the rest of the execution. Each time an event occurs, we select what type of event it is by sampling the uniform distribution using the probabilities above.

The advantage of our approach is that we have preserved the properties of the Poisson process and made it much easier to parallelize our code. We can use fixed timesteps and treat the events in each timestep independently. There is a 99.99% probability that only 1 event will happen at each time step, so we can treat our tiny event probabilities as entries in a fixed probability transition matrix. Moreover, each particle in our system can be treated independently at each step, allowing us to divide the computational work among many cores.

3 Software Design

3.1 Serial Workflow

Algorithm 1 describes the general serial algorithm.

3.2 Optimization and Parallelization Techniques

All of our simulation code is available on our public git repository for the course: https://github.com/ddeuel/CS205.

This was largely a compute focused project, as the only major data need was the storage of our position matrix. We used a hybrid model with distributed memory in MPI and shared memory in OpenACC. We divided our matrix among our eight nodes, and each node used GPU compute to state transformations using the transition probabilities and then update the position matrix. Finally, we used message passing between nodes to maintain the matrix globally and account for molecules diffusing across node divisions.

Our fast approach was possible because we carefully chose our timestep in such a way that we were able to guarantee strict locality. The probability of a particle moving between states more than once in a given timestep can be set to be arbitrarily close to 0 given sufficient computational resources. We chose 0.01% for practical reasons, but this could be reduced even further if more resources were available. Adjusting this probability did not measurably affect the outcome of our simulation with H molecules, but it might become important for simulations involving more complex interactions than

```
Data: grain size, time, temperature, binding energies, gas concentrations
Result: simulation of chemistry reactions on grain surfaces
calculate transition probabilities;
create position matrix P;
for t = 0 to T do
    for i = 1 to n^2 do
        if rand < tol then
            P(i) ← P(i)+desorbed molecule
        end
        if P(i) contains multiple molecules then
            if rand < tol then
                P(i) ← P(i) + products − reactants
            end
        end
    end
    for i = 1 to n^2 do
        if P(i) contains multiple molecules then
            for each molecule do
                if rand < tol then
                    s ← new state
                    P(i) ← P(i) − molecule
                    P(s) ← P(s) + molecule
                end
            end
        end
    end
end
```

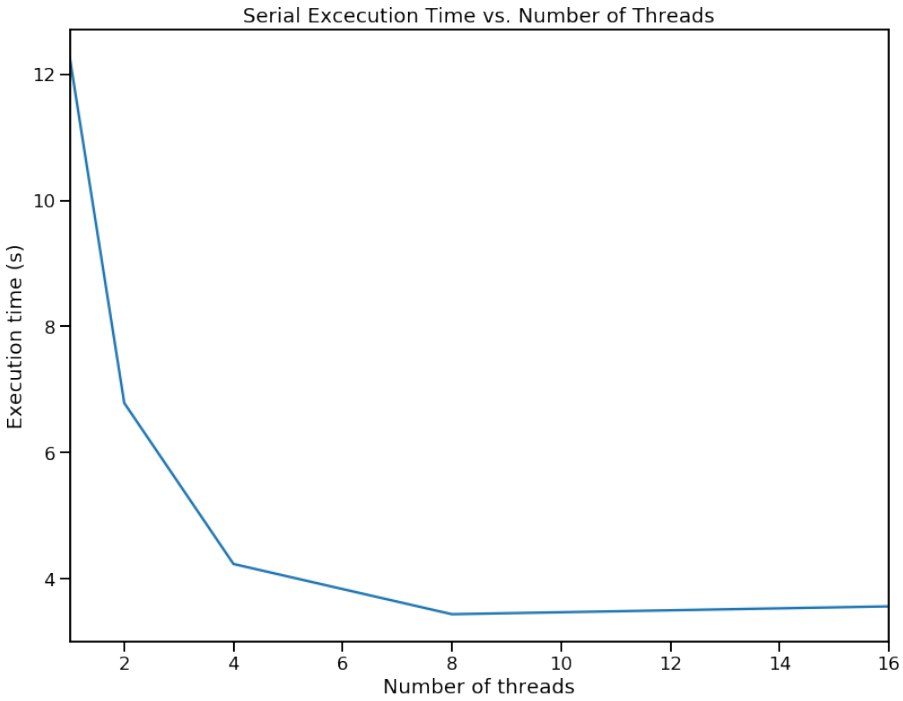
Algorithm 1: Serial algorithm for the grain-surface simulation.

those considered here, in which case we would set the value of this variable using a convergence test.

Locality has a huge effect on the resources required for our simulation. If we had implemented a full transition matrix and used matrix multiplication, the time complexity of our algorithm at each step would have been $O(n^6)$, where n is the number of available sites along each edge of a square grain surface. The total number of sites is n^2, the transition matrix would have been n^4, roughly a terabyte for our grain size, and a fully coated grain with n^2 particles would have required multiplications in $O(n^6)$. Instead, the locality guarantee allows us to iterate through an n^2 position matrix, treating each particle as independent and considering only its current site, the 8 adjacent diffusion sites, and desorption as possible transitions. In our final implementation, the time complexity at each step is then $O(n^2)$.

Since the number of particles on the grain surface can vary from 0 to n^2, we felt that a data structure with memory requirements proportional to the number of particles would create problems with load balancing. Instead, we opted for a position matrix n^2 elements, each of which stores the interaction energies of the particles at that site. Physically, the number of particles at each grain surface site is limited by space and the fact that particles in the same place will react with one another, so we were able to put an upper bound on the storage required for each matrix element. This fixed the size of our position matrix.

We took three approaches to parallelization on this code. The simplest approach was an OpenACC implementation of a single node multi-threaded CPU, shared memory parallel design. We decided to use OpenACC since this would allow for easy portability to a GPU parallel model with a simple flag change hopefully being the only thing required. Our file `grain_acc_cpu.c` uses an OpenACC pragma in our `hop()` function to parallelize the simulation of the diffusion and desportion processes on the grain. This process is represented in code as a nested for loop running over the occupancy matrix and executing probabilistic changes based on sampling to the grain matrix representation. We found that the CPU implementation was simplest to code because we could use the standard C `random()` functions to generate random numbers for sampling. We were able to achieve great scaling on this implementation with almost linear scaling as thread count increased, as seen below. We also achieved near linear speedup from the serial version of our code. There was overhead introduced through using OpenACC, as running with a single thread on OpenACC/PGI compilation and framework was significantly slower than running our serial version itself. This overhead based on pgprof analysis was largely due to fork, waits, and synchronization as we would expect. Using shared memory parallelization allowed us to not worry about separate copies of our matrix becoming out of sync. The scaling that we achieved with CPU parallelization can be seen in this figure where our speedup levels off with 8 threads:



The next parallel strategy was to then port our code over to be usable with GPU compute. This turned out to be less simple than simply removing the `-ta=multicore` CPU compliation tab for `pgcc` to compile. Instead we needed to rethink our data access methods. The standard C `random()` function is not supported during GPU computation. We attempted to use the Nvidia solution which is the `cuRAND` library. This didn't work despite an enormous amount of effort, so we moved to a new solution which was to precompute n^2 random values for each time step and access those random values in the GPU compute region by copying in the matrix. This added copy overhead that we would have liked to avoid, but we could not find a workable random number generator for the compute region. We used a similar data management strategy for the occupancy matrix. Initially we were copying in the occupancy matrix at every time step from the host device, but running pgprof we determined that the memory copy overhead was enormous and preventing good speedup results, as you can see in the screenshot below where the primary time use is on `[CUDA memcpy HtoD]`, host to device.

```
2.677715, ==74274== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   62.42%  53.293ms      4001  13.320us  1.2150us  51.232us  [CUDA memcpy HtoD]
                   28.78%  24.570ms      1000  24.569us  24.255us  25.984us  hop_89_gpu
                    6.84%  5.8372ms      1000  5.8370us  5.7600us  6.9440us  hop_124_gpu__red
                    1.97%  1.6795ms      1000  1.6790us  1.6630us  2.3680us  [CUDA memcpy DtoH]
      API calls:   36.33%  132.76ms         1  132.76ms  132.76ms  132.76ms  cuDevicePrimaryCtxRetain
                   17.95%  65.594ms      4001  16.394us  2.5440us  246.19us  cuStreamSynchronize
                   12.79%  46.726ms         1  46.726ms  46.726ms  46.726ms  cuDevicePrimaryCtxRelease
                    9.20%  33.633ms      1000  33.632us  17.145us  206.45us  cuMemcpyDtoHAsync
                    8.14%  29.728ms         1  29.728ms  29.728ms  29.728ms  cuMemHostAlloc
                    7.27%  26.574ms      4001  6.6410us  4.4830us  29.209us  cuMemcpyHtoDAsync
                    5.26%  19.234ms      2000  9.6170us  6.3330us  496.28us  cuLaunchKernel
                    2.69%  9.8449ms         1  9.8449ms  9.8449ms  9.8449ms  cuMemFreeHost
                    0.17%  609.37us         1  609.37us  609.37us  609.37us  cuMemAllocHost
                    0.08%  296.73us         7  42.389us  5.3090us  130.02us  cuMemAlloc
                    0.05%  165.26us         1  165.26us  165.26us  165.26us  cuModuleLoadData
                    0.03%  96.383us        20  4.8190us  4.5010us  5.3650us  cuEventSynchronize
                    0.02%  83.400us        21  3.9710us  3.4050us  4.5650us  cuEventRecord
                    0.00%  11.895us         1  11.895us  11.895us  11.895us  cuStreamCreate
                    0.00%  9.2110us         3  3.0700us     667ns  7.0400us  cuCtxSetCurrent
                    0.00%  7.7240us         4  1.9310us  1.0950us  2.8730us  cuPointerGetAttributes
                    0.00%  4.0550us         1  4.0550us  4.0550us  4.0550us  cuEventCreate
                    0.00%  3.9340us         3  1.3110us     513ns  2.4720us  cuDeviceGetCount
                    0.00%  3.6100us         1  3.6100us  3.6100us  3.6100us  cuDeviceGetPCIBusId
                    0.00%  2.9570us         4     739ns     568ns     929ns  cuDeviceGetAttribute
                    0.00%  2.7240us         2  1.3620us     675ns  2.0490us  cuModuleGetFunction
                    0.00%  2.2960us         2  1.1480us     592ns  1.7040us  cuDeviceGet
                    0.00%     825ns         1     825ns     825ns     825ns  cuCtxGetCurrent
                    0.00%     773ns         1     773ns     773ns     773ns  cuDriverGetVersion
                    0.00%     681ns         1     681ns     681ns     681ns  cuDeviceComputeCapability
```

We then decided to use an additional OpenACC pragma to copyin the occupancy matrix initially and maintain a local copy on the GPU memory instead of copying back and forth from the host device at every time step. This cut down the overhead dramatically and gave us good results. The new pgprof output screen can be seen below where the cuda memcpy is still a signficant time use, but is far less dramatic than before.

```
==82825== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   68.08%  24.391ms      1000  24.391us  24.063us  25.728us  hop_89_gpu
                   16.21%  5.8081ms      1000  5.8080us  5.7590us  6.5920us  hop_126_gpu__red
                   11.07%  3.9661ms      3001  1.3210us  1.2150us  51.230us  [CUDA memcpy HtoD]
                    4.64%  1.6640ms      1000  1.6630us  1.6310us  2.3680us  [CUDA memcpy DtoH]
      API calls:   43.57%  131.16ms         1  131.16ms  131.16ms  131.16ms  cuDevicePrimaryCtxRetain
                   16.39%  49.334ms         1  49.334ms  49.334ms  49.334ms  cuDevicePrimaryCtxRelease
                   12.29%  36.986ms      1000  36.985us  17.738us  536.69us  cuMemcpyDtoHAsync
                    9.55%  28.751ms         1  28.751ms  28.751ms  28.751ms  cuMemHostAlloc
                    5.00%  15.042ms      2000  7.5210us  5.4870us  498.40us  cuLaunchKernel
                    4.95%  14.910ms      3001  4.9680us  4.1120us  22.686us  cuMemcpyHtoDAsync
                    4.20%  12.633ms      3001  4.2090us  2.5220us  180.25us  cuStreamSynchronize
                    3.69%  11.114ms         1  11.114ms  11.114ms  11.114ms  cuMemFreeHost
                    0.20%  609.92us         1  609.92us  609.92us  609.92us  cuMemAllocHost
                    0.09%  271.38us         6  45.230us  5.4300us  127.13us  cuMemAlloc
                    0.05%  162.05us         1  162.05us  162.05us  162.05us  cuModuleLoadData
                    0.00%  10.611us         1  10.611us  10.611us  10.611us  cuStreamCreate
                    0.00%  9.2290us         3  3.0760us     699ns  7.1150us  cuCtxSetCurrent
                    0.00%  5.6810us         3  1.8930us     980ns  2.3550us  cuPointerGetAttributes
                    0.00%  3.7780us         1  3.7780us  3.7780us  3.7780us  cuDeviceGetPCIBusId
                    0.00%  3.1280us         4     782ns     682ns     899ns  cuDeviceGetAttribute
                    0.00%  2.9770us         3     992ns     496ns  1.7080us  cuDeviceGetCount
                    0.00%  2.3640us         2  1.1820us     572ns  1.7920us  cuModuleGetFunction
                    0.00%  2.0400us         2  1.0200us     573ns  1.4670us  cuDeviceGet
                    0.00%     847ns         1     847ns     847ns     847ns  cuDriverGetVersion
                    0.00%     739ns         1     739ns     739ns     739ns  cuDeviceComputeCapability
                    0.00%     698ns         1     698ns     698ns     698ns  cuCtxGetCurrent
0.569745,  OpenACC (excl):   26.69%  38.749ms      1000  38.748us  19.555us  538.57us  acc_enqueue_download@grain_acc.c:89
                   19.99%  29.022ms         1  29.022ms  29.022ms  29.022ms  acc_enter_data@grain_acc.c:168
                   13.30%  19.302ms      3000  6.4330us  5.4820us  29.064us  acc_enqueue_upload@grain_acc.c:89
                   12.01%  17.442ms      3000  5.8130us  3.8380us  503.11us  acc_wait@grain_acc.c:89
                    7.68%  11.155ms      1000  11.154us  9.7550us  501.11us  acc_enqueue_launch@grain_acc.c:89 (hop_89_gpu)
                    6.29%  9.1332ms      2000  4.5660us  1.3700us  100.14us  acc_enter_data@grain_acc.c:89
                    5.50%  7.9822ms      1000  7.9820us  7.2720us  34.787us  acc_enqueue_launch@grain_acc.c:89 (hop_126_gpu__red)
                    4.28%  6.2130ms      1000  6.2130us  5.6890us  119.00us  acc_compute_construct@grain_acc.c:89
                    4.05%  5.8810ms      2000  2.9400us  1.1450us  23.140us  acc_exit_data@grain_acc.c:89
                    0.14%  207.17us         1  207.17us  207.17us  207.17us  acc_device_init@grain_acc.c:168
                    0.04%  56.802us         1  56.802us  56.802us  56.802us  acc_wait@grain_acc.c:168
                    0.02%  30.592us         1  30.592us  30.592us  30.592us  acc_enqueue_upload@grain_acc.c:168
                    0.00%  2.6720us         1  2.6720us  2.6720us  2.6720us  acc_exit_data@grain_acc.c:168
                    0.00%       0ns      2000       0ns       0ns       0ns  acc_create@grain_acc.c:89
                    0.00%       0ns         1       0ns       0ns       0ns  acc_alloc@grain_acc.c:168
                    0.00%       0ns         1       0ns       0ns       0ns  acc_create@grain_acc.c:168
                    0.00%       0ns         1       0ns       0ns       0ns  acc_delete@grain_acc.c:191
                    0.00%       0ns         2       0ns       0ns       0ns  acc_alloc@grain_acc.c:89
                    0.00%       0ns      2000       0ns       0ns       0ns  acc_delete@grain_acc.c:134
```

Our GPU speedup was similar to our CPU speedup which was unexpected. We would've hoped for even better speedup on the GPU since we already saw the workload was highly parallelizable, but the overhead of memory copy to the host, and the relatively small problem size meant that we achieved only similar results to the much simpler CPU based implementation.

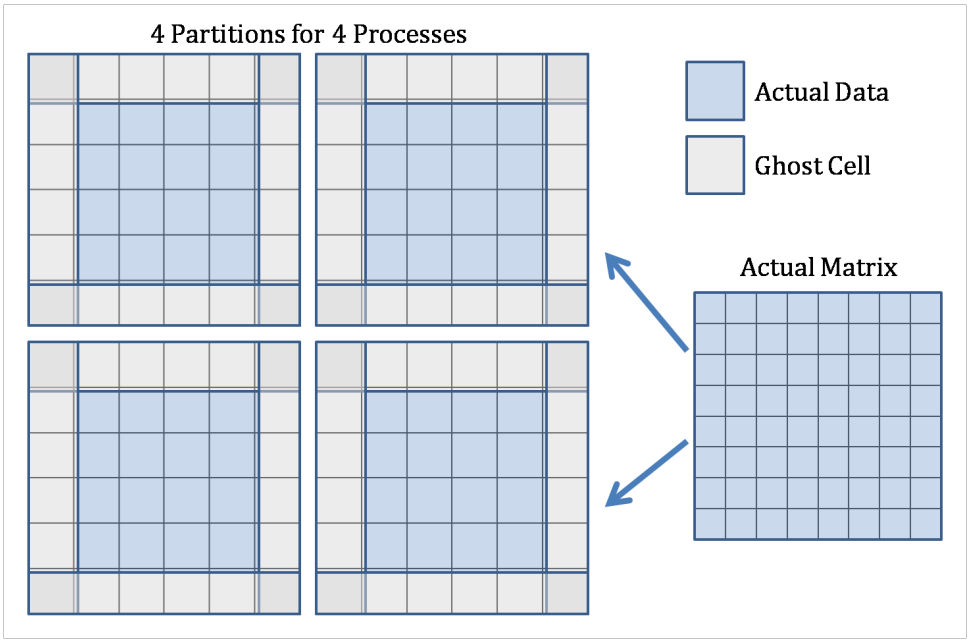


Figure 2: Illustration of our partitioning approach to parallelization using MPI

Our final approach was parallelization over distributed memory using MPI. In this approach, we partitioned the simulated grain surface over multiple processors, so that the resulting submatrices could be operated upon simultaneously. This was a particularly challenging approach, for several reasons: (1) the particles had to be able to traverse the entire grain, which meant that we had to facilitate communication (i.e., message-passing) across submatrices; (2) since the particles could roll over the grain edge and appear on the other side, we had to choose the moments of message-passing carefully, to avoid conflicts from overlapping diffusion; and (3) we had to employ both local and global techniques to update the submatrices at each timestep and then synchronize their results at the end of the simulation.

We used ghost cells as the medium of communication across the edges of submatrices. Figure 2 illustrates this approach. In summary, we partitioned the square grain surface among a given number of processors (restricted so that the grain surface could be split evenly in both length and width among the processors). We then padded the top, bottom, left, and right of each submatrix with ghost blocks, where no actual data was stored. At each timestep, each processor scanned the submatrix for particles and any diffusion or desorption events; whenever the processors reached the second-to-last or last cell within a row representing actual data, then the edges of each submatrix were updated by the ghost cells.

3.3 System Setup

All code was written in C. We used a g3.4xlarge instance for serial, CPU, and GPU timings to keep the timings as consistent as possible for cross referencing speedup. We followed the infrastructure guide i5 on OpenACC on AWS provided by the course for instance configuration and compiler set up except that we used the latest 19.4 version of PGI.

To configure the shell environment to use OpenACC and compile with pgcc run these commands must be run on each new instance shell session:

```
export PGI=/opt/pgi;

export PATH=/opt/pgi/linux86-64/19.4/bin:$PATH;
```

The compilation lines used for generating executables was:

```
serial: gcc -DUSE_CLOCK grain.c timing.c -o grain -lm

multi-threaded CPU: pgcc -Minfo -ta=multicore -DUSE_CLOCK grain_acc_cpu.c
timing.c -o grain_acc_cpu -lm

GPU: pgcc -acc -Minfo -DUSE_CLOCK grain_acc.c timing.c -o grain_acc -lm
```

To change the number of threads used in CPU parallel execution set the environment variable like this:

```
export ACC_NUM_CORES=16
```

To execute we have a format of:

```
./executable n  NUM_STEPS TIMING_ONLY_FLAG
```

Where `n` is an integer value for the grain size as `n x n` , `NUM_STEPS` is the integer number of time steps, and `TIMING_ONLY_FLAG` is a integer of 0 or 1 for 0 as verbose output, and 1 as timing value output only.

An example run input would then be:

```
./grain_acc 300  100000 0
```

For distributed-memory parallelization over MPI, we used the t2.2xlarge AWS instance(s). We compiled and ran the code as follows:

```
serial compiler: mpicc -DUSE_CLOCK grain_serial.c timing.c -o
grain_serial -lm

parallel compiler: mpicc -DUSE_CLOCK grainMPI.c timing.c -o
grain_serial -lm

serial: mpirun -np 1 ./grain_serial size numpsteps 0

parallel (4 processors, 1 node): mpirun -np 4 ./grainMPI size numsteps 0

parallel (4 processors, MPI cluster with 2 nodes): mpirun -np 4 -hosts
main,node1 ./grainMPI size numsteps 0

parallel (4 processors, MPI cluster with 3 nodes): mpirun -np 4 -hosts
main,node1,node2 ./grainMPI size numsteps 0
```

Where `grain_serial` and !grainMPI.c ! are the serial and parallel files, *size* is the grain size, *numsteps* is the number of steps, 0 is a flag that prints out a summary of the performance, and *main*, *node*1, and *node*2 were the arbitrary names given to the nodes in the cluster.

When running our serial implementation, timings showed exponential growth with the size of our grain. This is to be expected as we are operating on a square matrix resulting in at least $O(n^2)$ time complexity.

Timing our code when we executed in in serial, we found that for the serial portions of our code took 0.32 seconds. The parallelizable portions took 9.55 seconds. The fraction, p, of parallelizeable runtime to total runtime is then 0.968. According to Amdahl's law, this means that our theoretical maximum speedup S_{max} for a fixed problem size is 31.25 times.

$$S_{max} = \frac{1}{1-p} = \frac{1}{0.032}$$

In practice, we our found the speedups for a fixed problem size to max out at approximately 16 times. The data and figures associated with our speedup and results can be found in the performance evaluation section.

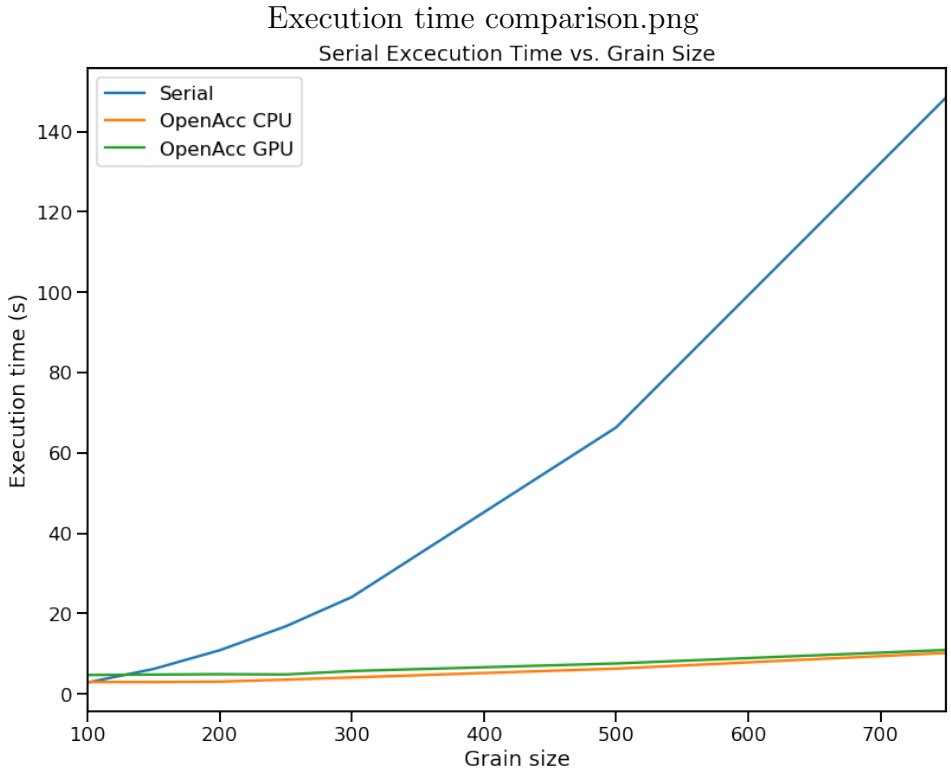


Figure 3: Serial execution times of our surface simulations for square grains with n sites on each edge, or n^2 sites in total

4 Discussion

4.1 Performance Evaluation

Previously, we hypothesized that each time step would consume roughly the same amount of resources so that the overall execution time would be proportional to the runtime of the individual timesteps, $O(n^2)$. Execution times are plotted in 3 as a function of the side length of the grain, n. In practice, we observed the following weak scaling speedups as we increased our problem size:

Compared to the serial case, parallelization over distributed memory led to significant improvement in performance. Figure 5 shows the speed-up of the simulation as a function of problem size for four processors on a single MPI node. Figure 6 shows the effect on the simulation speed-up given by distributing processors over a cluster of MPI nodes. Figure 7 demonstrates the overhead associated with synchronizing and outputting the results from each submatrix. We see that the speed-up relative to the serial case for the simulation is quite significant, and is roughly constant with problem size. Distributing the work over multiple nodes in a cluster does not drastically alter performance for the simulation alone. However, the overhead associated with post-simulation synchronization and output for multiple nodes is not negligible. Notably this overhead is most detrimental to performance for small problem sizes, and decreases in significance as problem size increases.

Serial	OpenAcc CPU	OpenAcc GPU	Grain size
2.832046	2.969	4.702021	100
6.208168	2.947	4.808163	150
10.86386	3.055	4.904208	200
16.822279	3.554	4.84202	250
24.115229	4.1	5.694505	300
66.341999	6.294	7.574742	500
148.588536	10.197	10.941632	750

Figure 4: Serial execution times of our surface simulations for square grains with n sites on each edge, or n^2 sites in total

4.2 Future Directions

While we have made vast steps in decreasing the computational and temporal costs of our simulation, there is still much work that can be done. Firstly, our codes can be improved for better user-flexibility. For example, currently the number of processors that can be used for our MPI approach is restricted, such that the grain surface can be split evenly in both length and width across the processors. Relaxing this restriction would require an even more complex ghost-cell and message-passing implementation, but would be worthwhile in exchange for a wider variety in the number of processors that can be used. Our simulation also currently requires that the grain surface representation be square in shape; relaxing this restriction would allow more complicated lattice or grid geometries, which could more accurately represent actual dust grains in space.

There is also much to be done to make the results of our simulation scientifically relevant and significant. Currently we simulate a baseline grain-surface scenario: a smooth grain surface in a gas containing only hydrogen atoms. In reality, the gas could contain other atoms and molecules, such as carbon monoxide or oxygen, depending on the local environment (e.g., diffuse interstellar medium vs. a dense molecular cloud). Incorporating more atoms/molecules into our simulation would allow us to study the formation of more complex and interesting chemistry (e.g., water) in different environments. Grain surfaces in space are also likely not smooth, but typically rough and mixed, with possible regions of charge. Expanding our simulation to be able to generate inhomogeneous surfaces, with surface and binding energies that vary across the sites, would allow us to determine how grain-surface chemistry depends on these traits.

References

Sarah Ballard. Predicted Number, Multiplicity, and Orbital Dynamics of TESS M-dwarf Exoplanets. , 157(3):113, Mar 2019. doi: 10.3847/1538-3881/aaf477.

J. K. Blitzstein and J. Hwang. *Introduction to Probability.* CRC Press, Taylor Francis Group, 2015.

Q. Chang, H. M. Cuppen, and E. Herbst. Continuous-time random-walk simulation of

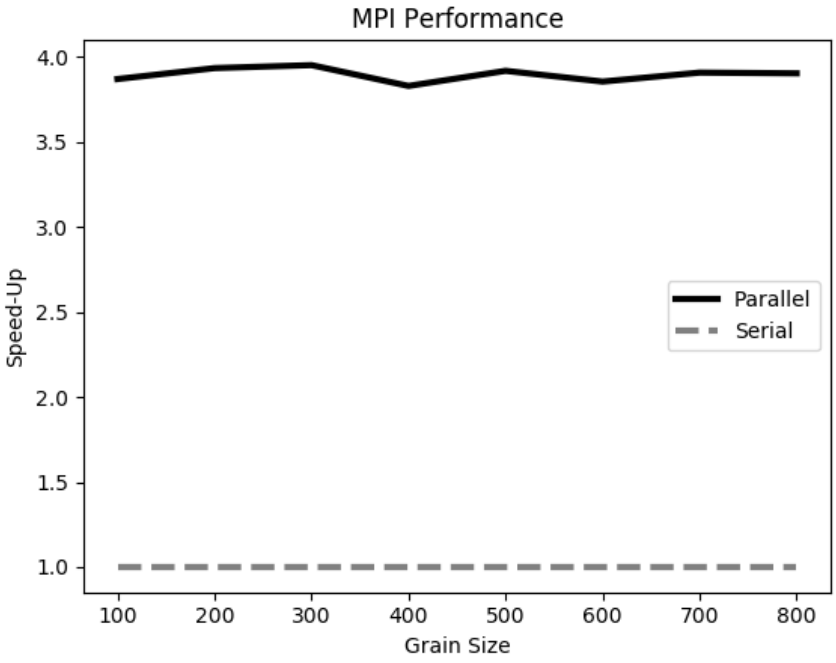


Figure 5: Parallel vs. serial speed-up for the simulation using four processors in MPI

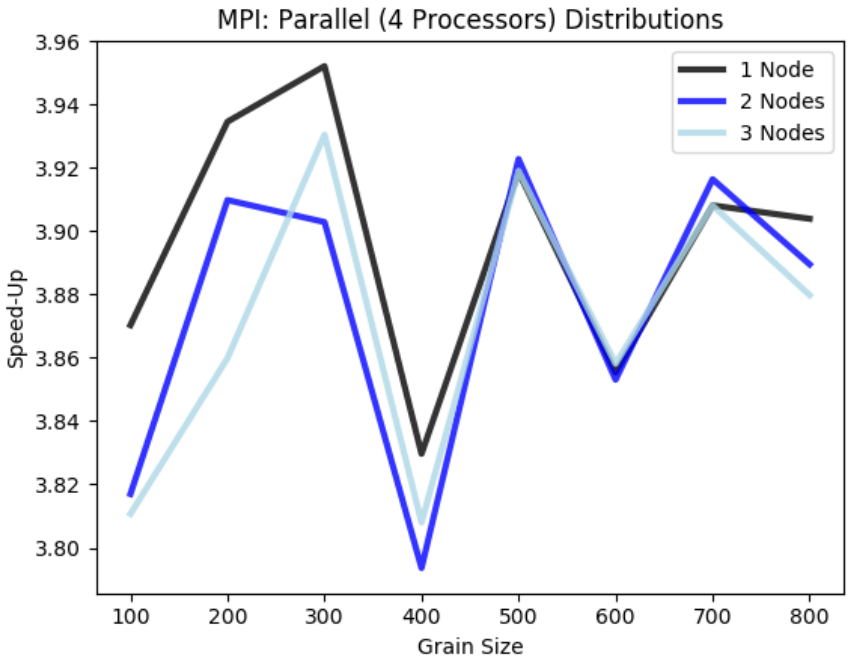


Figure 6: Parallel speed-up for the simulation using four processors in an MPI cluster

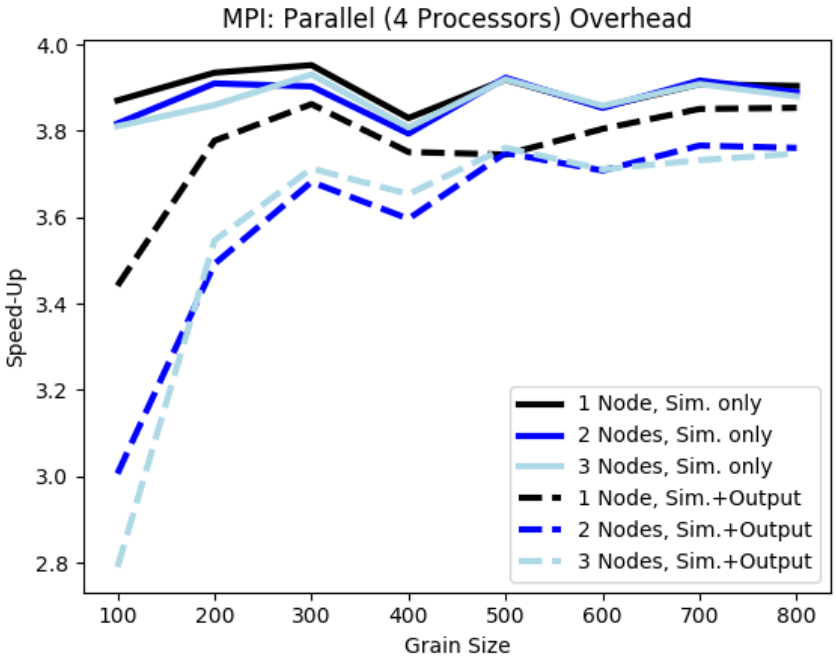


Figure 7: Overhead associated with synchronizing and outputting results across the different submatrices

H_2 formation on interstellar grains. , 434(2):599–611, May 2005. doi: 10.1051/0004-6361:20041842.

Qiang Chang and Eric Herbst. Interstellar Simulations Using a Unified Microscopic-Macroscopic Monte Carlo Model with a Full Gas-Grain Network Including Bulk Diffusion in Ice Mantles. , 787(2):135, Jun 2014. doi: 10.1088/0004-637X/787/2/135.

H. M. Cuppen, C. Walsh, T. Lamberts, D. Semenov, R. T. Garrod, E. M. Penteado, and S. Ioppolo. Grain Surface Models and Data for Astrochemistry. , 212(1-2):1–58, Oct 2017. doi: 10.1007/s11214-016-0319-3.

George R. Ricker, Roland Kraft Vanderspek, David W. Latham, and Joshua N. Winn. The Transiting Exoplanet Survey Satellite Mission. In *American Astronomical Society Meeting Abstracts #224*, volume 224 of *American Astronomical Society Meeting Abstracts*, page 113.02, Jun 2014.

Valentine Wakelam, Emeric Bron, Stephanie Cazaux, Francois Dulieu, Cécile Gry, Pierre Guillard, Emilie Habart, Liv Hornekær, Sabine Morisset, Gunnar Nyman, Valerio Pirronello, Stephen D. Price, Valeska Valdivia, Gianfranco Vidali, and Naoki Watanabe. H_2 formation on interstellar dust grains: The viewpoints of theory, experiments, models and observations. *Molecular Astrophysics*, 9:1–36, Dec 2017. doi: 10.1016/j.molap.2017.11.001.