CS267: Homework 2-3

Yarden Goraly Chris Lai Alec Li

1 Introduction

Simulation is a key application of parallel computing, widely used in fields such as physics, robotics, and weather forecasting. These applications rely on accurately simulating interactions between objects and often require processing vast amounts of data in real time. In this work, we use Graphical Processing Units (GPUs) with CUDA programming to parallelize a simple particle simulator in which particles move within a box, interacting with both the walls and neighboring particles. The main challenge in this work was using an appropriate data structure for effective parallelization. We ended up using a particle-binning approach in which a array of particles are sorted by their respective bin (corresponding to cell in a grid). We found that particle simulation performance for high numbers of particles was much better than other parallelization techniques we used in previous homeworks for this task.

2 Implementation and Design

The general structure mainly is inspired by what the recitation had. Structurally, the algorithms are similar, having a linear force apply phase, moving phase, counting, a prefix sum, and then a reorganizing step, in which we later figured out that using a variation of counting sort led to the best results.

2.1 Initial implementation

In our initial implementation, we build off of the idea of our serial implementations from previous parts of this homework, but with some key modifications to prepare for GPU optimizations. For instance, we still partition our particles into uniform grids, but instead of using a vector of vectors, we use four global arrays to keep track of not only the particles themselves, but also their positions in an array sorted by the bins. These four arrays are shown in Listing 1.

Listing 1 Initial Global Arrays

```
// number of particles per bin
int* cpu_bin_counts;
// starting index for each bin, as a prefix sum of `bin_counts`
int* cpu_prefix_sum;
// Particle Indices Array (Mapped to original particle struct)
int* particle_indices;
// Key for sorting
int* particle_bin_ids;
```

All these arrays are populated with appropriate values at the start of the simulation. The processes of assigning a bin ID to each particle and counting the number of particles in each bin are parallelized using CUDA kernels. For each kernel in this project, we set the block size (number of particles per thread) to

$$\left\lceil \frac{\texttt{num\_particles}}{\texttt{TOTAL\_NUMBER\_THREADS}} \right\rceil,$$

and we set the number of threads to 256.

Listing 2 Initial CUDA Particle Simulation Algorithm

```
1:  procedure InitSimulation(parts, num_parts, size)
2:      blks ← ⌈num_parts / NUMBER_THREADS⌉
3:      num_bins_along_axis ← ⌈size / CELL_SIZE⌉
4:      Allocate memory for bin_counts, prefix_sum, particle_indices, particle_bin_ids
5:      Initialize particle_indices ← [0, 1, ..., num_parts − 1]
6:      Assign each particle to its corresponding bin ID
7:      Count particles in each bin
8:      Compute prefix sum of bin counts
9:      Sort particles by bin ID
10: end procedure
11: procedure SimulateOneStep(parts, num_parts, size)
12:     Compute forces using binning structure          ▷ Using custom CUDA kernel
13:     Move particles and update bin IDs               ▷ Using custom CUDA kernel
14:     Reset and recount particles in each bin         ▷ Using custom CUDA kernel
15:     Recompute prefix sum of bin counts              ▷ Using thrust::exclusive_scan
16:     Sort particles by bin ID                        ▷ Using thrust::sort_by_key
17: end procedure
```

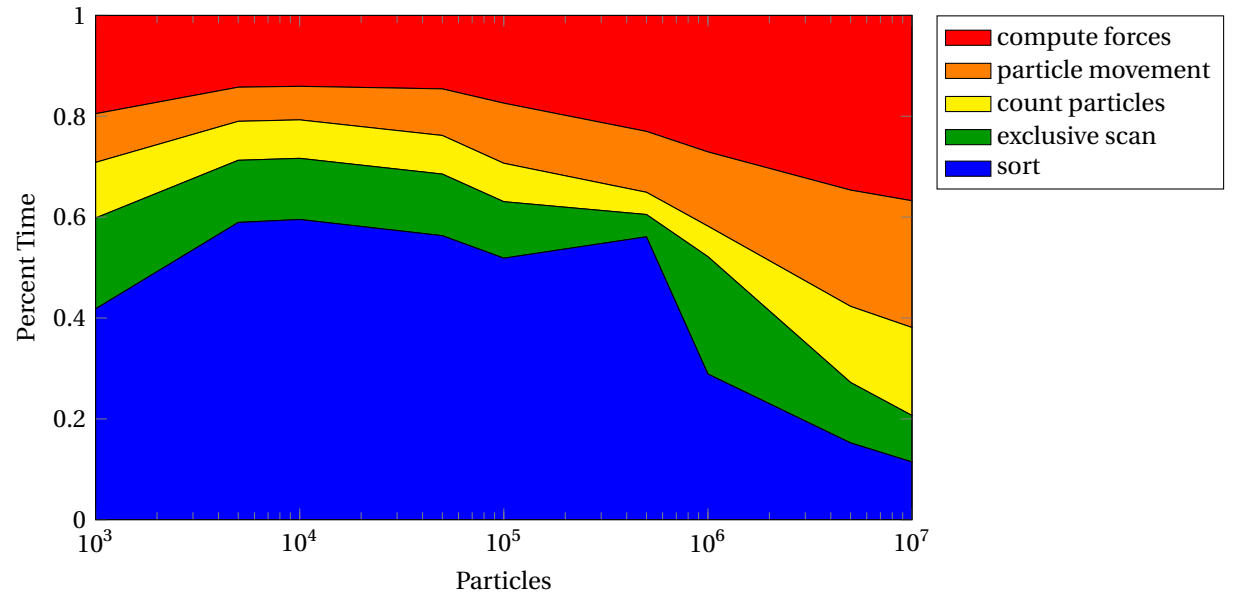
This approach takes 11.9044 s for 10 million particles, which is not bad for an initial implementation. The breakdown of the compute time for this initial implementation is shown in Table 1.

Table 1: Time breakdown for the initial implementation, for 10^7 particles

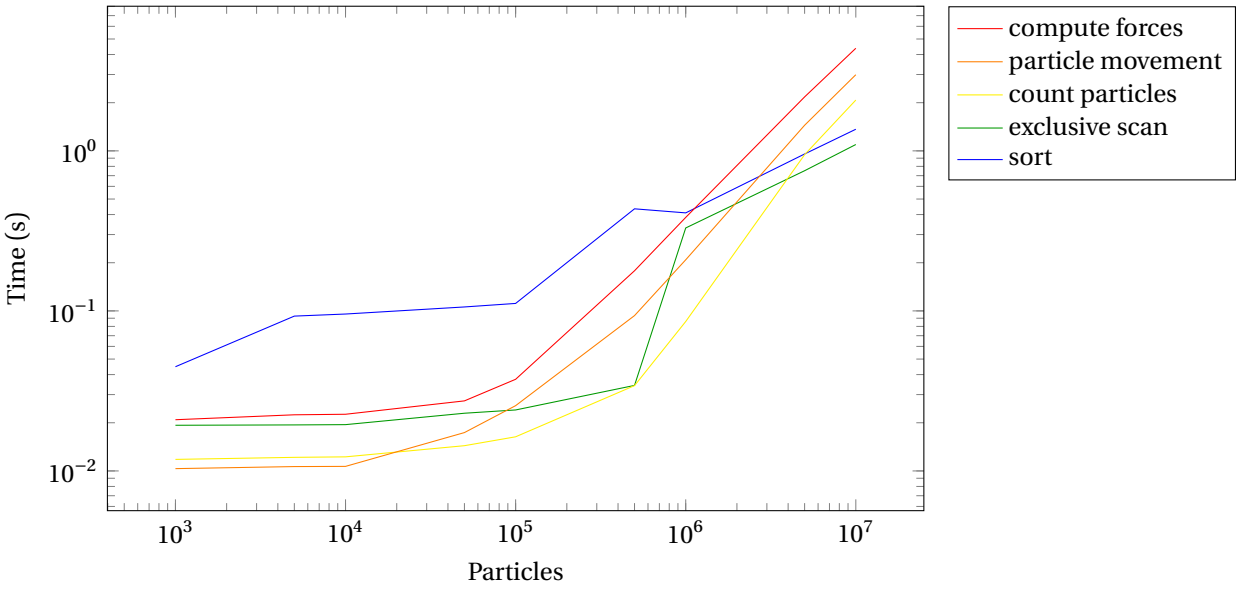
Phase	Time (s)
Compute forces	4.375845
Move particles	2.991125
Count particles in bins	2.076845
Exclusive scan	1.09562
Sort	1.36498
Total compute time	11.904415

Within each step of the simulation, we perform the following tasks: compute the forces between neighboring particles, move the particles, then update the state of bins. So far, this approach is identical to a serial version of this task. In order to parallelize our approach using CUDA, we used three kernels. The first one computes neighboring forces for each particle. The second moves each particle according to its appropriate force. Finally, the third kernel recounts the particles for each bin. Notably, this same function was used for initializing the particles. Once this is done, we regenerate `prefix_sum` and sort `particle_indices`, effectively updating the particles to their new bins after moving.

Figure 1a breaks down the initial implementation's computation time in our GPU implementation by phase. Here, we can see that for a small number of particles, sorting takes the majority of the time, while for a larger number of particles, other phases like the force computation and the data movement to the GPU takes more and more time. The absolute time taken in each phase is shown in Fig. 1b.



(a) Stacked area chart depicting the proportional time taken in each phase



(b) Line chart depicting the absolute simulation time for each phase

Figure 1: Breakdown of runtime by phase for the initial implementation in Section 2.1

This aligns with our intuition—as the number of particles increases, the force computation will take longer, and the particle position updates would take longer as well. On the other hand, sorting is generally very efficient, and Thrust has highly parallelized implementations that allow for the sorting to scale well. This means that sorting will take a smaller slice of the simulation time as the number of particles increases, as everything else starts to dominate.

2.2 Synchronization description

In our implementation we used a few strategies to ensure synchronization. First, when incrementing bin counts we use `atomicAdd` rather than incrementing naively. We do this to prevent race conditions as the atomic add ensures that each bin is incremented by at most one thread at a time. Another way we ensured synchronization

is by designing our kernels such that computing forces, moving particles, and bin bucketing are all performed separately, with each one acting as a barrier for the threads. Since each of these steps depend on the previous step, we assured that each step is parallelized on the GPU and that each thread finishes before moving on to the next step. In addition, we designed our data structures such that each kernel either reads or writes to the same array, which prevents data races.

2.3 Optimizations

We'll now discuss some additional optimizations we implemented, which mainly aim to mitigate the weaknesses of the initial implementation. In particular, we introduced packing to promote memory coalescing in the NVIDIA GPUs, as well as the usage of a custom counting sort to replace `thrust::sort_by_key`.

2.3.1 Packing, converting array of structs to structs of arrays

In an effort to improve compute time, which was the main bottleneck of our initial implementation, we try to utilize NVIDIA Ampere GPU's memory coalescing feature, which allows for quick simultaneous memory reads and writes for threads in a warp. In order to guarantee that, the access of individual attributes of a struct (ex. xy position, velocity, etc.) must happen at the same time, accessing contiguous memory addresses.

To do so, we modify how we store the particles; in the initial implementation, we store an array of particle structs, while now we store a struct of arrays, with one array for each attribute. The struct is defined in Listing 3.

Listing 3 Struct of Arrays for Particle Attributes

```
struct ParticleSOA {
    double* x;
    double* y;
    double* vx;
    double* vy;
    double* ax;
    double* ay;
    int* bin_ids;
    int* particle_indices;
};
```

In this struct, we include the 6 original fields of the particle struct. In addition, we also keep the bin ids to keep track of which bin each particle is in, alonside the particle's index, which is used to map the values in the arrays back into the original struct. It should be noted that the particle indices would only be required in a simulation where there are occasional checks to follow a particular particle's trajectory (ex. for output for the correctness check).

After this introduction of our structure of arrays, in the `init_simulation` function, we initialize all eight of these arrays by allocating memory, and then copying all the structure's information into the array of structs. The bin IDs are initialized with the xy positions, and the `particle_indices` array is arbitrarily initiated with an arbitrary sequence using `thrust::sequence`. After that, all of the functions are re-factored such that read and write operations are distinct instructions. An example is shown in Listing 4.

In this copy from our Struct of Arrays (SOA) to Arrays of Structs (AOS), we are first reading in the positional values in a coalesced manner. We ensure this by also sorting all of the arrays, as mentioned, by permutation once the particle indices array is sorted. This yields an approximate 20% speedup in compute time across the first few phases, reducing the time for data access significantly.

With this modification, we successfully decreased the compute time all over the board; however, a new bottleneck emerged: the sorting of the particles. This is because in this new implementation, in order to ensure that memory coalescing still occurs for read and writes in the CUDA kernels, we have to make sure that all of the arrays are sorted—this adds an overhead to permute all of the attributes in the struct of arrays, after an initial `sort_with_key` of the particle indices based on `bin_ids`. The breakdown of compute time is shown in Table 2.

Listing 4 Memory Coalescing example in Custom Copy CUDA Kernel Example

```
__global__ void soa_to_aos(ParticleSOA soa, particle_t* aos, int num_parts) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    // --- snip ---

    // These reads are coalesced (good)
    double x = soa.x[tid];
    double y = soa.y[tid];
    ...

    // Write to the appropriate location in AOS (scattered writes)
    aos[particle_index].x = x;
    aos[particle_index].y = y;
    ...
}
```

Table 2: Time breakdown after restructuring, for 10^7 particles

Phase	Time (s)
Compute forces	1.66139
Move particles	0.740445
Count particles in bins	0.654696
Exclusive scan	0.769473
Sort	7.68582
Total compute time	11.5132
Repacking	3.421
Simulation time	15.7854

Note that there is an additional 3.421 s of overhead in each step, to repack the particle's individual attributes (i.e. `x`, `y`, `vx`, `vy`, `ax`, `ay`) back into the original structs. Later, we realized that we only need to copy the x and y positions back into the structs for the correctness check, which reduced this time down to 1.2 s.

2.3.2 $O(N)$ bin reorganizing using a counting sort

This optimization primarily aims to target the bottleneck of the sorting algorithm as a particle reorganization mechanism in our algorithm. The initial sort that we have been using all this while, implemented with `sort_with_keys`, with our benchmarks, took 7 s. Additionally, because of repacking and copying overhead, we really need to reduce the compute time there in order for the repacking's results to reap it's computational benefits.

We perform bin reorganizing as a variation of counting sort with CUDA Kernels, and the algorithm is as follows:

Listing 5 Counting Sort as a Reorganizing Algorithm in CUDA

```
1:  procedure BinBucketingKernel(particles, prefix_sum, bin_counters)
2:      tid ← threadIdx.x + blockIdx.x × blockDim.x
3:      if tid < num_particles then
4:          bin_id ← particles.bin_ids[tid]
5:          bin_pos ← atomicAdd(bin_counters[bin_id], 1)
6:          dst_idx ← prefix_sum[bin_id] + bin_pos
7:          x_temp[dst_idx] ← particles.x[tid]              ▷ Move particle data to new location
8:          y_temp[dst_idx] ← particles.y[tid]
9:          indices_temp[dst_idx] ← particles.particle_indices[tid]
10:         ...                                             ▷ Copy remaining particle properties
11:     end if
12: end procedure
```

This successfully reduces the runtime complexity of the reorganization portion from $O(N\log(N))$ to $O(N)$, providing us tremendous speedup in re-organizing all of the array of structs (AOS). Other than that, as mentioned in the previous section, we also reduced the random-access write to just the position of the particles, essentially reducing our repacking cost to just 1.74 s.

Table 3: Time breakdown after counting sort bin reorganizing

Phase	Time (s)
Compute forces	1.52961
Move and count particles	1.02111
Exclusive scan	0.681396
Bin bucketing	2.21718
Repacking	1.74437
Simulation time	7.2165

This optimization ended up with a final simulation time of 7.2165 s for 10^7 particles, and it is our final submission for the solution for this assignment. A breakdown of the compute times are shown in Table 3, and additional details are discussed in Section 3.

3 Results

Figure 2 compares the performance of various parallelization implementations. The serial and GPU starter code performs the worst, as expected. OpenMP and MPI are similar in speed, as an order of magnitude faster.

Our GPU implementation is another order of magnitude faster, but only for a larger amount of particles. For a smaller amount of particles, the overhead of the algorithm starts to be significant, causing some slowdowns. However, the GPU code scales well with the number of particles—the simulation time is approximately constant for up to 10^5 particles, and slowly increases afterward.

Figure 3a breaks down the computation time in our GPU implementation by phase. Here, we can see that the time is split fairly consistently as the number of particles increases; the fraction of time spent in most phases stays the same. The largest variation comes with the repacking, which takes an increasing fraction of time as the number of particles increases. It is interesting to note the sudden increase in exclusive scan time at 10^6 particles; this is likely due to the additional memory usage for 10^6 particles, which may have introduced some additional overheads with the exclusive scan.

This behavior aligns with our intuition, especially as we look at the absolute time taken for each phase in Fig. 3b. Most notably, here, we can see that the time spent generally increases jointly across all phases, meaning that

there isn't any particular phase that dominates the computation (which is notably different to the behavior for an earlier iteration in Fig. 1, where sorting and force computation dominate). This suggests that we're at a point where further optimizations may only impact the scaling by small amounts—we'd need to jointly optimize *all* phases in order to have any larger impact on the overall simulation time (otherwise, if we only optimize a specific phase, the other phases will still dominate the overall simulation time).

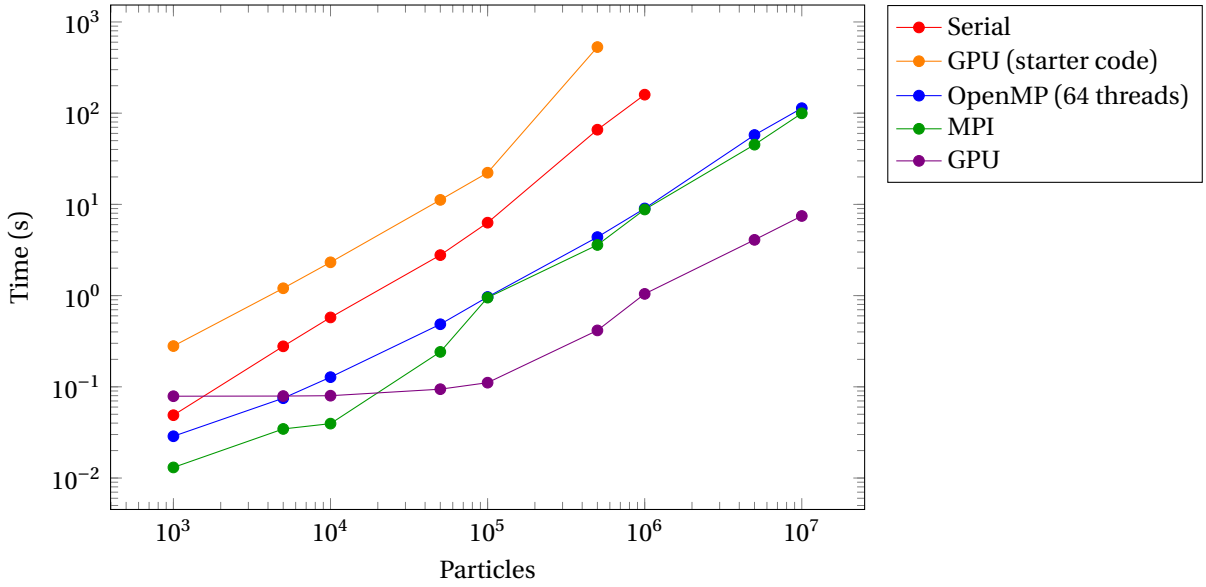
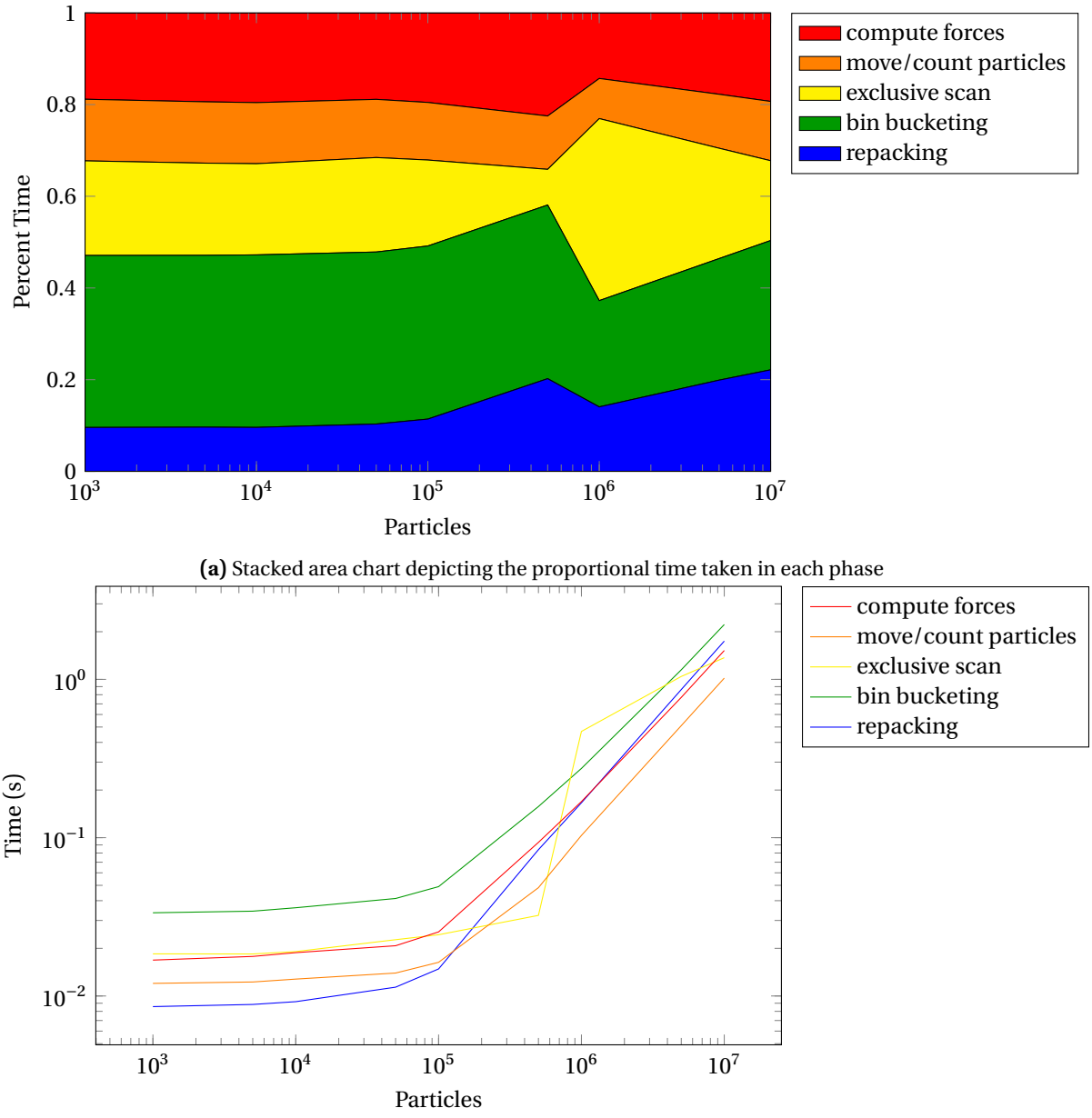


Figure 2: Comparison of simulation time for various particle counts and parallelization methods

4 Contributions

Yarden, Chris, and Alec all collaborated equally. Alec worked on the initial implementation of the algorithm, creating custom CUDA kernels for the various phases of the process and working out nuances of using CUDA in C++; he also worked on benchmarking the code at various points in the development progress, and created the plots in the writeup. Yarden worked on designing and implementing an initial implementation of the particle binning algorithm and worked on parallelizing it using CUDA kernels. Chris worked with Yarden to flesh out the details of the initial algorithm, and worked with Alec to implement the initial algorithm; additionally, he worked on exploring and implementing all of the other optimizations to speed up the algorithm.



(a) Stacked area chart depicting the proportional time taken in each phase

(b) Line chart depicting the absolute simulation time for each phase

Figure 3: Final breakdown of runtime by phase, after all optimizations described in Section 2.3