

Wind Tunnel Simulation

Cicio Ionuț

<https://github.com/CuriousCI/wind>

Note: to compile in DEBUG mode, the `inline` keyword must be removed from the functions. The report is lengthy because the project is very big and it took a lot of effort, tries and errors.

14/02/2025

1 Wind

The software simulates, in a simplified way, how the air pressure spreads in a Wind Tunnel, with **fixed obstacles** and **flying particles**. [...] All particles have a mass and a resistance to the **flow air**, partially **blocking** and **deviating** it.

— `handout.pdf`, page 1 Introduction

The sequential code has two main parallelization targets: **flow updates** and **interactions with particles**; the latter is a big obstacle to parallelization for a number of reasons:

- **accesses to memory** depend on the positions of the particles
- the `move_particle()` function is very expensive and requires a **copy of the air flow**
- air flow positions with particles don't need to be always updated
- **some particles move**, can end up on top of each other and in different sections of the tunnel, making it very hard to distribute the workload and avoid **race conditions**

The next few pages are dedicated to analyzing the techniques used in all the implementations. These optimizations not only lead to a better scale-up relative to the sequential code, but also better efficiency when increasing the number of processes / threads.

1.1 The effects of particles on performance

From this point on fixed obstacles will be referred to as **fixed particles**, and flying particles as **moving particles** to be consistent with the code.

1.1.1 Take what you need, leave what you don't

In the sequential code, a particle (either moving or fixed) is described by a `struct Particle` s.t.

```
typedef struct {
    unsigned char extra;
    int pos_row, pos_col; ①
    int mass; ②
    int resistance; ③
    int speed_row, speed_col; ④
    int old_flow; ⑤
} particle_t; // Particle → particle_t to fit my code style
```

Listing 1: `wind.c`

This way of storing particles is very costly in terms of memory usage: not all particles need all the attributes, and not all attributes are needed everywhere. In fact:

- the only attributes fixed particles need are the position ① and the resistance ③, and these never change
- moving particles need the position ①, the speed ④ and the mass ② only when moving, the rest of the time, the position ① is enough
- no particle actually needs the extra field and the old_flow ⑤

To solve this problem the idea is to partially use the “*struct of arrays* instead of *array of structs*” strategy:

```
typedef struct { int row, col; } vec2_t; ①
typedef struct { vec2_t pos, speed; } particle_m_t; ②
```

Listing 2: util.h

The particle_m_t struct ② contains the attributes needed by moving particles, and is way smaller than particle_t.

```
particle_m_t *particles_m = ...; ③
vec2_t *particles_pos = ...;
vec2_t *particles_m_pos = particles_pos + num_particles_f; ④
int *particles_res = ...;
int *particles_back = ...;
int *particles_m_mass = ...;
```

Listing 3: wind_omp_2.c

This way the attributes are separated in different arrays. It’s interesting to note that the positions of moving particles are stored twice:

- in particles_m ③
- in particles_m_pos ④

That’s because the position in particles_m is multiplied by a certain PRECISION, and the actual position inside the matrix is the one in particles_pos, having two advantages:

- the actual position inside the matrix can be pre-calculated for the fixed particles
- the data that needs to be transferred for moving particles is just a vec2_t, half the size of a particle_m_t and 28% of a particle_t ($\frac{64B}{232B}$)

Using multiple arrays benefits each implementation in different ways.

1.1.2 Bringing order in a world of chaos

The particles are **generated randomly across the matrix**, this means that the probability of **cache misses** increases drastically. This problem is easy enough to handle: the particles need to be sorted by position.

```

qsort(particles, num_particles_f, ... , particle_cmp); ①
qsort(
    particles + num_particles_f,
    num_particles_m, ... ,
    particle_cmp
); ②

```

Listing 4: qsort is a standard C library function works in $O(n \log(n))$

The array with the particles is divided in two parts: on the left the fixed particles ①, and on the right the moving particles ②. To improve the performance the best bet is to sort each part individually, because:

- each type of particle is treated differently, it's more efficient to have moving particles close to each other when processing them
- if the sorting was across all the particles, moving particles would quickly mess up the order

Due to the movement patterns of moving particles, separating the types of particles greatly improves the cache hit percentage. The moving particles tend to move towards the bottom of the matrix and stay there (see Figure 1, the moving particles are marked by []). After some iterations, all the accesses to the matrix when working with moving particles are close to each other.

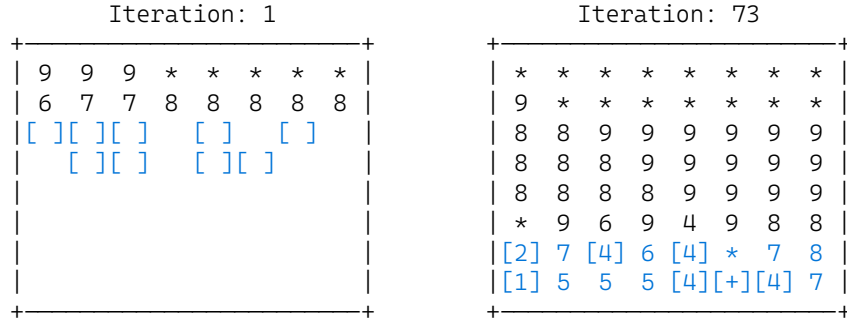


Figure 1: particles movement pattern from top to bottom

It's useful to sort only once at the beginning, there is no benefit in doing it again later:

- fixed particles don't move, so there's no reason to sort them again
- moving particles don't need to be sorted due to the movement patterns in Figure 1 (*I tried different strategies for sorting multiple times during the execution, but there weren't any speed-up improvements*)

To stress the importance of sorting, this is the output of `perf stat` on OpenMP in different sorting conditions.

```

perf stat -d ./wind_omp 500 500 1000 0.5 0 500 1 499 1.0 1 499
1.0 1902 9019 2901

```

```
// No sorting
L1-dcache-load-misses:u      27.29% of all L1-dcache accesses
// Sorting only fixed particles
L1-dcache-load-misses:u      13.19% of all L1-dcache accesses
// Sorting only moving particles
L1-dcache-load-misses:u      17.01% of all L1-dcache accesses
// Sorting all types of particles
L1-dcache-load-misses:u      3.48% of all L1-dcache accesses
```

1.1.3 Computational cost

Adding particles to the simulation is very expensive: not only the `move_particle()` function takes a lot of time (4.111s, Figure 2) and it's very hard to optimize, but the workload of the `main()` function increases from ≈ 60 ms to 1.360s due to all the processing that must be done on the particles (updating the flow on the positions marked by particles, calculating the pressure of each particle and changing the flow around each particle by adding the pressure of the particle).

The graphs below aren't representative of the most optimal solution (particles aren't sorted, particles' attributes aren't split into multiple arrays etc...), but it's a good enough representation of what happens.





Function / Call Stack	CPU Time ▼	Clockticks
<code>move_particle</code>	4.111s 	15,573,600,000
<code>main</code>	1.370s 	5,182,800,000
<code>update_flow</code>	0.451s 	1,786,400,000
<code>__memmove_avx_unaligned_erms</code>	0.039s 	100,800,000

Figure 2: Intel Vtune, “Microarchitecture Exploration” analysis of `wind_seq` with a tunnel full of particles on input

512 512 1000 0.5 0 512 1 511 1.0 1 511 1.0 4556 7307 1911



Function / Call Stack	CPU Time ▼	Clockticks
<code>main</code>	59.829ms 	238,000,000
<code>__memmove_avx_unaligned_erms</code>	32.906ms 	120,400,000

Figure 3: Intel Vtune, “Microarchitecture Exploration” analysis of `wind_seq` without any particles on input

512 512 10000 0.5 0 512 1 511 0.0 1 511 0.0 4556 7307 1911

The same measurements were made with `valgrind --tool=callgrind` and the results are similar (for `callgrind` I compiled with `-ggdb3` and without `-O3` to have a more detailed breakdown of each line's cost).

For this reason measurements are presented for both a situation without particles and a situation with a lot of particles (for some implementations the efficiency and scale-up are way better when there aren't any particles).

1.2 OpenMP

1.2.1 Wave propagation implementation

```
int last_wave = iter + 1 < rows ? iter + 1 : rows;
#pragma omp parallel for reduction(max : max_var)
for (int wave = wave_front; wave < last_wave; wave += STEPS) {
    for (int col = 0; col < columns; col++) {
        /* do stuff... */
    }
} /* End propagation */
```

I want to discuss this section briefly because it has a nested loop. I tried everything: collapsing the loop, using a different types of scheduling (static with different sizes, dynamic with different sizes, guided), nothing improves the efficiency of the basic pragma... this is because **waves** are distant (each wave is 8 rows apart), so collapsing the loops can cause inefficient memory accesses (it's better for a thread to handle the whole row, instead of assigning different parts of it to different threads). This solution is bad when the input has very few rows and a lot of columns.

1.2.2 Particles interactions implementation

Many of the for-loops that work on particles are separated.

```
if (num_particles_m > 0)
    #pragma omp for nowait
    for (int p = 0; p < num_particles_m; p++)
        update_flow(particles_pos[num_particles_f + p]);

if (num_particles_f > 0)
    #pragma omp for
    for (int p = 0; p < num_particles_f; p++)
        update_flow(particles_pos[p]);
```

Listing 5: wind_omp.c

```
+-----+
|           [ ] [F][M]           |
|           [ ][ ][ ] [ ] [ ]    |
|           [ ][ ][ ][ ]         |
|           [ ][ ]               |
|           [ ]                 |
|[ ]                               |
|           [ ][ ][ ]           |
|           [ ] [ ] [ ][ ]       |
|           [ ][ ][ ]           |
|           [ ] [ ] [ ][ ]       |
|[ ][ ] [ ] [ ][ ] [ ]          |
+-----+
```

This works very well because fixed and moving particles are sorted separately. If the loops were merged, it could happen that two different threads would be assigned particles close to eachother (e.g. fixed particle [F] to thread 1 and moving particle [M] to thread 3 in Listing 5), leading to **false sharing** when updating the flow.

There are two parts of the code that must be sequential. The first one is the inlet flow update (Listing 7) because it uses `erand48()` ① which is `MT-Unsafe`.

```
for (j = inlet_pos; j < inlet_pos + inlet_size; j++) {
    double noise = 0.5 - erand48(random_seq); ①
    accessMat(flow, 0, j) = ... (pressure_level + noise);
}
```

Listing 7: Inlet flow update

The other part is when the resistance of the particles changes the flow.

```
#pragma omp single
for (int p = 0; p < num_particles_m; p++) {
    int row = ..., col = ..., back = ...;
    accessMat(flow, row, col) -= particles_m_back[p]; ①
    accessMat(flow, row - 1, col) -= particles_m_back[p]; ①
    accessMat(flow, row - 1, col - 1) -= particles_m_back[p]; ①
    accessMat(flow, row - 1, col + 1) -= particles_m_back[p]; ①
}
```

This part is problematic because multiple particles can overlap, so changes to the same position ① by different particles must be atomic. The compiler uses **vectorized instructions**, which are really fast. So fast, in fact, that none of the other methods I tried is more efficient:

- `#pragma omp atomic` is slower because make the instruction atomic and prevents the compiler from optimizing
- I tried `omp_lock_t` in two different ways:
 - by creating a matrix of locks, to lock each required cell individually, but the overhead to use locks was too big
 - the second idea was to lock entire rows (to reduce the overhead), but this also proved to be inefficient
- `#pragma omp reduction()` didn't work either, because reducing big arrays in OpenMP can break **very easily** the **stack limit**
- I tried doing something similar to a reduction manually by using `#pragma omp threadprivate` and allocating each thread's section on the heap, but this also proved inefficient

There's nothing that can be done for **moving particles**.

Function / Call Stack	CPU Time ▼
move_particle	3.375s
main._omp_fn.2	1.074s
func@0x23510	0.589s
update_flow	0.199s
main._omp_fn.3	0.080s
func@0x23370	0.060s
main._omp_fn.5	0.048s
__memmove_avx_unaligned_erms	0.039s
particle_cmp	0.031s

Figure 4: Intel Vtune, “Microarchitecture Exploration analysis” on wind_omp with a tunnel full of particles

Nonetheless, this part can be parallelized *very efficiently without race conditions* for **fixed particles** (see the code in wind_omp.c, lines 656–720 and 857–888). The idea is to distribute the fixed particles among the threads in such a way that all particles that have the same position are assigned to the same thread. Then particles on the border between two threads must be removed, and handled later.

1.2.3 Notes on sorting

Sorting the particles at the beginning is quite expensive, so I decided to parallelize this part too (even if it’s done only once).

```
#pragma omp parallel
{
    #pragma omp single
    {
        if (num_particles_f > 0)
            #pragma omp task
            qsort(particles, num_particles_f, ...);

        if (num_particles_m > 0)
            #pragma omp task
            qsort(particles_moving, num_particles_m, ...);
    }
}
```

Listing 8: wind_omp.c / wind_mpi_omp.c / wind_omp_cuda.cu sorting

The simplest solution I found was to create 2 tasks, so fixed and moving particles can be sorted at the same time... this solution is quick, but it doesn’t scale with more than 2 threads. A better solution, enough time provided, would be to implement a merge sort, or use other solutions like [this one](#) that uses the standard qsort, or a [bitonic sort](#).

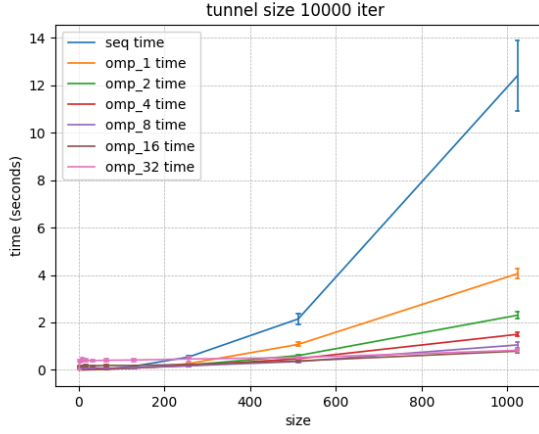


Figure 5: x axis tunnel size, no particles

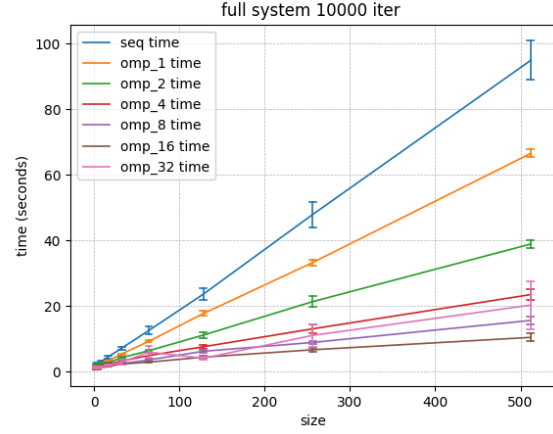


Figure 6: x axis particles band size

speed-up				
	tunnel size			
program	128	256	512	1024
omp_1	2.16	2.06	2.0	3.06
omp_2	2.13	2.68	3.57	5.39
omp_4	1.96	3.06	4.56	8.29
omp_8	1.56	3.17	6.11	11.82
omp_16	0.82	2.41	5.74	15.72
omp_32	0.37	1.18	4.13	15.03

speed-up				
	particles band size			
program	64	128	256	512
omp_1	1.35	1.33	1.44	1.43
omp_2	1.95	2.11	2.24	2.44
omp_4	2.53	3.09	3.65	4.03
omp_8	3.44	3.77	5.35	6.05
omp_16	4.23	5.27	7.12	9.02
omp_32	2.08	5.7	4.31	4.66

efficiency				
	tunnel size			
program	128	256	512	1024
omp_1	2.16	2.06	2.0	3.06
omp_2	1.06	1.34	1.79	2.69
omp_4	0.49	0.76	1.14	2.07
omp_8	0.2	0.4	0.76	1.48
omp_16	0.05	0.15	0.36	0.98
omp_32	0.01	0.04	0.13	0.47

efficiency				
	particles band size			
program	64	128	256	512
omp_1	1.35	1.33	1.44	1.43
omp_2	0.98	1.06	1.12	1.22
omp_4	0.63	0.77	0.91	1.01
omp_8	0.43	0.47	0.67	0.76
omp_16	0.26	0.33	0.44	0.56
omp_32	0.07	0.18	0.13	0.15

This data looks strange, because the efficiency in some cases (e.g. omp_1, omp_2) is greater than 1. This happens on other implementations too, because the sequential code does a lot of useless/inefficient work:

- it always copies the flow matrix
- it always resets the particle_locations
- it always recalculates the position inside the matrix of **all** particles
- it doesn't sort the particles
- it doesn't take advantage of how fixed and moving particles are separated

1.3 CUDA

1.3.1 Implementation

The implementation requires 4 kernels, 3 of which behave like the respective sequential code.

```
__global__ void move_particles_kernel()
__global__ void update_particles_flow_kernel()
__global__ void update_back_kernel()
__global__ void propagate_waves_kernel()
```

The interesting one is `propagate_waves_kernel()`, which can be optimized by bringing the row above the section of the current wave assigned to the block into **shared memory** (including the halo region).

```
__shared__ int temp[WAVE_BLOCK_SIZE + 2];
int s_idx = threadIdx.x + 1;
temp[s_idx] = accessMat(d_flow, wave - 1, col);
if (col > 0)
    temp[s_idx - 1] = accessMat(d_flow, wave - 1, col - 1);
if (col < columns - 1)
    temp[s_idx + 1] = accessMat(d_flow, wave - 1, col + 1);
__syncthreads();
```

The CUDA version is very fast because all copies from host to device are made at the beginning, only 1 variable needs to be moved from the device to the host each iteration and a single row must be copied from host to device every STEPS iterations.

1.3.2 Speed-up

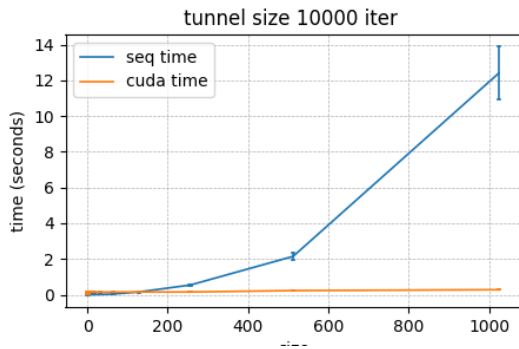


Figure 7: x axis tunnel size, no particles

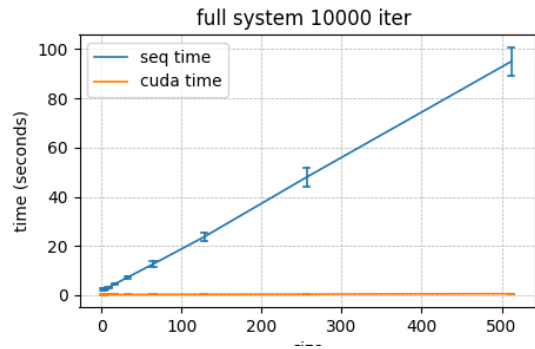


Figure 8: x axis particles band size

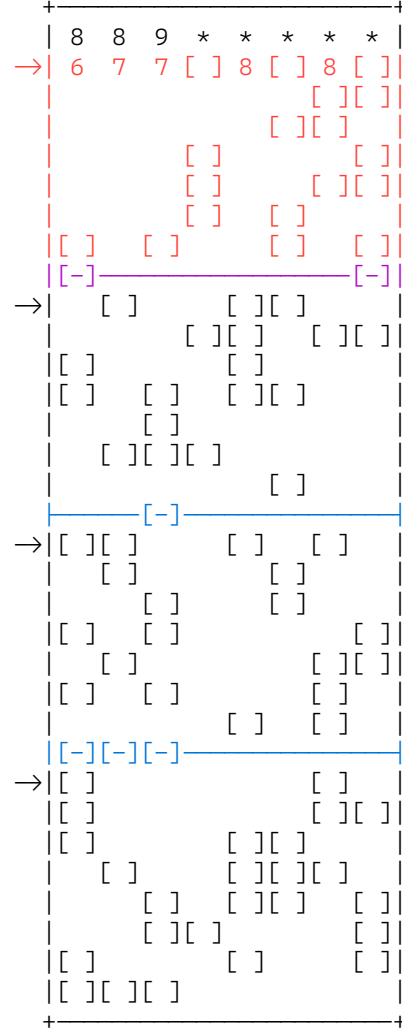
speed-up				
	tunnel size			
	128	256	512	1024
cuda	1.02	3.58	9.38	44.75

speed-up				
	particles band size			
	64	128	256	512
cuda	43.37	70.32	117.37	167.83

1.4 MPI

The idea around the MPI implementation is to assign each process some consecutive **sections** of the tunnel (a section is a slice of STEPS rows, like the one in red/purple), because each iteration only one row in each section is changed. At the end of each STEPS iterations, each process sends to its successor the last row it modified (in blue/purple), so the successor can calculate the first row by using the last row of the predecessor.

This works very well when there are no particles in the input. When there are particles, every STEPS iterations an `MPI_Allgatherv()` is called on the sections of the matrix, so each process can move the **moving particles** assigned to it. After moving all the particles, process 0 gathers with `MPI_Gatherv` the new positions of the particles (by using a **custom datatype** `MPI_VEC2_T`), and does all the processing on the particles (updates the flow around the particles etc...). Here the separation of the position comes handy, because it's faster to transfer less data.



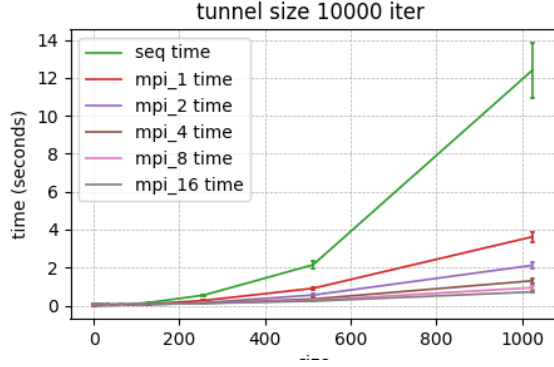


Figure 9: x axis tunnel size, no particles

speed-up				
	tunnel size			
program	128	256	512	1024
mpi_1	1.88	2.07	2.37	3.42
mpi_2	3.42	3.63	3.9	5.86
mpi_4	2.73	4.53	6.29	9.52
mpi_8	2.04	4.65	7.95	13.19
mpi_16	1.72	4.81	9.34	17.33

efficiency				
	tunnel size			
program	128	256	512	1024
mpi_1	1.88	2.07	2.37	3.42
mpi_2	1.71	1.82	1.95	2.93
mpi_4	0.68	1.13	1.57	2.38
mpi_8	0.25	0.58	0.99	1.65
mpi_16	0.11	0.3	0.58	1.08

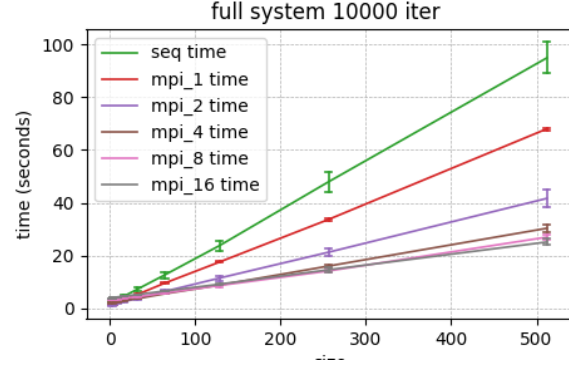


Figure 10: x axis particles band size

speed-up				
	particles band size			
program	64	128	256	512
mpi_1	1.33	1.36	1.42	1.4
mpi_2	1.98	2.08	2.26	2.28
mpi_4	2.31	2.72	3.0	3.13
mpi_8	2.19	2.76	3.4	3.53
mpi_16	1.94	2.61	3.29	3.78

efficiency				
	particles band size			
program	64	128	256	512
mpi_1	1.33	1.36	1.42	1.4
mpi_2	0.99	1.04	1.13	1.14
mpi_4	0.58	0.68	0.75	0.78
mpi_8	0.27	0.34	0.42	0.44
mpi_16	0.12	0.16	0.21	0.24

The same problem with the efficiency that OpenMP had arises here too, for the same reasons. It's interesting to note how, when particles are added, the implementation is way less scalable, because process 0 does a lot of the work alone and it can't be distributed very easily. When there aren't any particles the implementation is **very weakly scalable** and **less strongly scalable**. When there are particles, it scales very badly.

1.5 MPI + OpenMP

The MPI+OpenMP version doesn't add very much: it has the same division in sections like the MPI version, and moving particles are distributed equally among processes. OpenMP parallelizes the movement of the particles in each process, and parallelizes the interactions with the particles for process 0.

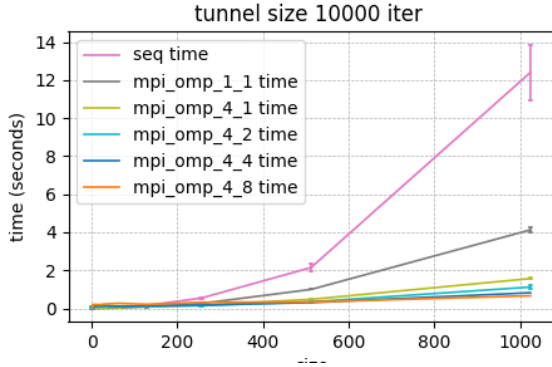


Figure 11: x axis tunnel size, no particles

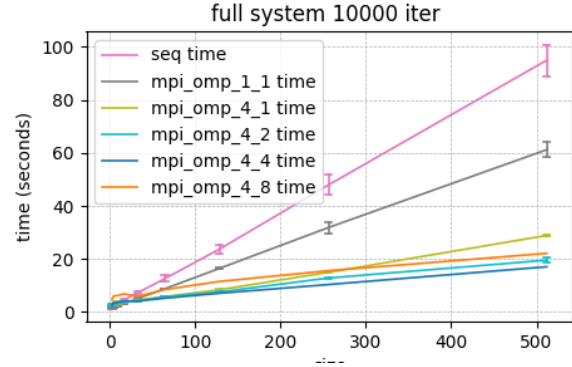


Figure 12: x axis particles band size

speed-up				
	tunnel size			
program	128	256	512	1024
mpi_omp_2_2	1.74	3.74	3.91	8.28
mpi_omp_2_4	1.54	3.65	5.84	11.46
mpi_omp_2_8	1.17	2.82	8.05	15.71
mpi_omp_4_2	1.54	3.8	6.35	11.09
mpi_omp_4_4	1.16	3.12	7.02	15.13
mpi_omp_4_8	0.72	1.71	6.39	18.76

speed-up				
	particles band size			
program	64	128	256	512
mpi_omp_2_2	2.4	2.99	3.21	3.55
mpi_omp_2_4	2.55	3.47	4.7	4.65
mpi_omp_2_8	3.27	3.67	5.18	5.88
mpi_omp_4_2	2.21	3.13	3.73	4.85
mpi_omp_4_4	2.41	3.34	4.62	5.56
mpi_omp_4_8	1.49	2.05	3.07	4.29

efficiency				
	tunnel size			
program	128	256	512	1024
mpi_omp_2_2	0.43	0.93	0.98	2.07
mpi_omp_2_4	0.19	0.46	0.73	1.43
mpi_omp_2_8	0.07	0.18	0.5	0.98
mpi_omp_4_2	0.19	0.48	0.79	1.39
mpi_omp_4_4	0.07	0.19	0.44	0.95
mpi_omp_4_8	0.02	0.05	0.2	0.59

efficiency				
	particles band size			
program	64	128	256	512
mpi_omp_2_2	0.6	0.75	0.8	0.89
mpi_omp_2_4	0.32	0.43	0.59	0.58
mpi_omp_2_8	0.2	0.23	0.32	0.37
mpi_omp_4_2	0.28	0.39	0.47	0.61
mpi_omp_4_4	0.15	0.21	0.29	0.35
mpi_omp_4_8	0.05	0.06	0.1	0.13

1.6 OpenMP + CUDA

The only difference between the the OpenMP + CUDA implementation and the simple CUDA implementation is the parallelization of the particles' pre-processing before copying the data from the host to device.

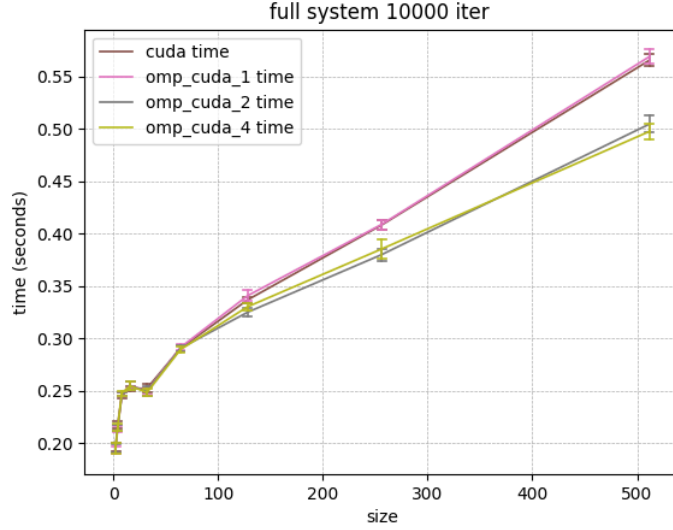


Figure 13: CUDA vs OpenMP+CUDA, x axis size of particles band

It doesn't scale much over 2 threads because the sorting isn't scalable, but I wrote it to to squeeze a bit more out of the CUDA implementation.

	speed-up			
	particles band size			
	64	128	256	512
omp_cuda_1	43.24	69.5	117.21	166.81
omp_cuda_2	43.36	72.87	126.1	188.1
omp_cuda_4	43.54	71.71	124.34	190.76

1.7 Other

In the arch folder other “*failed*” implementations / attempts can be found. I tried implementing a MPI + CUDA code too, but it proved to be very inefficient, due to the amount of data needed to be transferred to the GPU to do the calculations.