Wind Tunnel Simulation

Cicio Ionuț
https://github.com/CuriousCI/wind

10/02/2025

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: the **update of the air flow**, and the **interactions with the partilces**; the latter is a big obstacle to parallelization due to a few important reasons:

- the `move_particle()` function is very expensive and requires a copy of the matrix with the air flow
- accesses to memory depend on the positions of the particles
- air flow positions with particles don't need to be always updated
- **some** the particles move, which has both pros and cons:
  - pre-calculating the optimal way to access the memory is virtually impossible, but flying particles move close to eachother, increasing the cache hit percentage as the simulation goes

The next few pages are dedicated to analyzing the tricks used to optimize memory usage. These optimizations not only lead to a better scale-up realtive 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 partilces 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.

```c
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 was renamed to particle_t for
consistency
```

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 `pos_row,pos_col` ① and `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 = ... ; ④
int *particles_res = ... ,
    *particles_back = ... ,
    *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_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` ($\frac{64\text{B}}{232\text{B}}$).

1.1.2 Bringing order in a world of chaos

The particles are generated randomly across the matrix, this means that the probabilty of cache misses increases drastically if the matrix is bigger. This problem is easy enough to handle: the particles need to be sorted by position (first row then column).

```
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, first of all due to implementation reasons:

- for MPI it's easier to use `MPI_Gatherv` to gather the results of `move_particle()`
- for OpenMP: the workload of `move_particle()` is more evenly distributed and cache-friendly, and it makes possibile to parallelize parts which otherwise would have been sequential

Secondly, due to the movement patterns of the 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 Listing 5, the moving particles are marked by `[ ]`). After some iterations, all the accesses to the matrix when working with moving particles are close to eachother, without having to sort again.

```
      Iteration: 1                      Iteration: 73
+------------------------+      +------------------------+
| 9  9  9  *  *  *  *  * |      | *  *  *  *  *  *  *  * |
| 6  7  7  8  8  8  8  8 |      | 9  *  *  *  *  *  *  * |
|[ ][ ][ ]   [ ]   [ ]   |      | 8  8  9  9  9  9  9  9 |
|   [ ][ ]   [ ][ ]      |      | 8  8  8  9  9  9  9  9 |
|                        |      | 8  8  8  8  9  9  9  9 |
|                        |      | 8  8  8  8  8  9  9  9 |
|                        |      | 7  8  8  8  8  9  8  8 |
|                        |      | *  9  6  9  4  9  8  8 |
|                        |      |[2] 7 [4] 6 [4] *  7  8 |
|                        |      |[1] 5  5  5 [4][+][4] 7 |
+------------------------+      +------------------------+
```

Listing 5: particles movement pattern from top to bottom

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

- the fixed particles don't move, so there's no reason to sort them again
- the moving particles move closer to eachother until they overlap at the bottom, so there's no benefit in sorting them again

(I tried different strategies for sorting multiple times during the execution, but there weren't any speed-up improvements. I don't belive a distributed sorting algorithm would benefit either)

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

```
perf -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       21.88% of all L1-dcache accesses
// Sorting only moving particles
L1-dcache-load-misses       13.86% of all L1-dcache accesses
// Sorting all particles
L1-dcache-load-misses       2.85%  of all L1-dcache accesses
```

1.2 OpenMP

1.2.1 Implementation

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

```c
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 6: `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 6), 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 8) because it uses `erand48()` ① which is `MT-Unsafe`.

```c
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 8: Inlet flow update

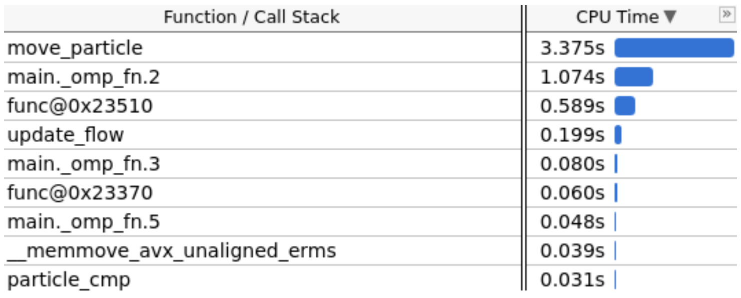
The other part is when the resistance of the particles changes the flow.

```c
#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. For this block the compiler uses **vectorized instructions**, which are really fast. So fast, in fact, that non of the methods I've tried can beat vectorized instructions:

- `#pragma omp atomic` not only is slower because make the instruction atomic, it also provents the compiler from using vectorized instructions in this context
- I tried using `omp_lock_t` in two different ways:
  - the first attempt was to create a matrix of locks, to lock each required cell individually, but the overhead to use locks was too big
  - the second attempt was to lock entire rows (to reduce the overhead for the locks by working with bigger sections), 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; even when setting the stack limit to unlimited the performance wasn't good
- 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 required some kind of sequential code at some point

There's nothing that can be done for **moving particles**, and it's not worth to try to parallelize this part, as most of the execution time is spent elsewhere (see Figure 1).



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 1: Intel Vtune analysis

Even so, this part can be parallelized *very efficiently* for **fixed particles**.

```
#pragma omp parallel
{
  int thread_num = omp_get_thread_num();
  for (int p = f_displs[thread_num]; ①
      p < f_displs[thread_num] ① + f_counts[thread_num]; ②
      p++) {
      int row = ... , col = ... , back = ... ;
      accessMat(flow, row, col) -= back;
      accessMat(flow, row - 1, col) += (back / 2);
      accessMat(flow, row - 1, col - 1) += (back / 4);
      accessMat(flow, row - 1, col + 1) += (back / 4);
}
```

The idea is to distribute all the fixed particles among the threads by pre-calculating the displacements ① and the counts ② in such a way that all particles that have the same position are assigned to the same thread, so no race conditions can happen. The fixed particles are already sorted, so it's enough to distribute the particles evenly among the threads, and check the particles on the borders.

1.2.2 Speed-up & efficiency

	Order of Tunnel				
	64	128	256	512	1024
seq	0.0354	0.1238	0.4659	1.7831	10.2625
omp_1	0.0242	0.0808	0.2976	1.1357	4.3416
omp_2	0.026	0.064	0.1765	0.6305	2.2364
omp_4	0.0389	0.0604	0.1398	0.3748	1.2126
omp_8	0.0635	0.0784	0.1253	0.3288	0.7496
omp_16	0.1459	0.1516	0.1805	0.2808	0.5982
omp_32	0.3169	0.3057	0.3279	0.4117	0.5811

Table 1: Run-times without any particles with 10000 iterations

1.3 CUDA

1.3.1 Implementation

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

```
__shared__ int temp[32 + 2];

int w = wave_front + blockIdx.y * 8, col = ... ;
if (/* not valid ... */) return;
```

```
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();
```

1.3.2 Speed-up & efficiency

1.4 MPI

1.5 MPI + OpenMP

2 Fails

2.1 Pthread

As the problem looked very complex at the beginnning and really hard to parallelize, I originally started working with `pthread`. While trying different strategies with `pthread` I found out ways to simplify the code, at the point I was able to actually use OpenMP to make significant improvements with way less effort.

2.2 OpenMP

2.3 CUDA