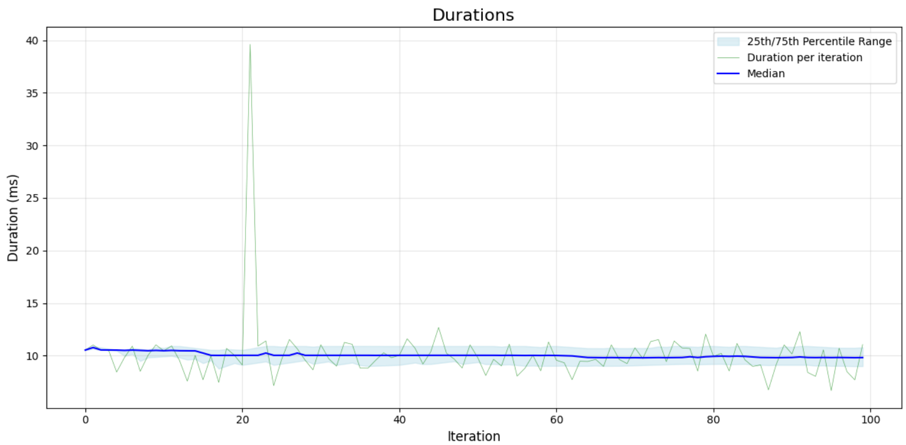
Parallelism ablation

Baseline

Durations for the baseline (no parallelism) implementation:



The median duration is 10 ms.

Parallelism

We implement parallelism as follows:

1. Use the `DestroyMethod` to generate a subset of size `subset_size * parallelism`, where `parallelism` is the amount of threads/processes to use.
2. Split the `flat_subset` into `parallelism` subsets of size `subset_size`.
3. Copy `PathTable` and `instance` to each worker thread/process.
4. Run standard iteration using `LowLevelSolver`.
5. Get lowest `new_num_collisions` from worker threads.
6. If it is better than the current `num_collision`, update the solver's `path_table` and `instance` attributes.

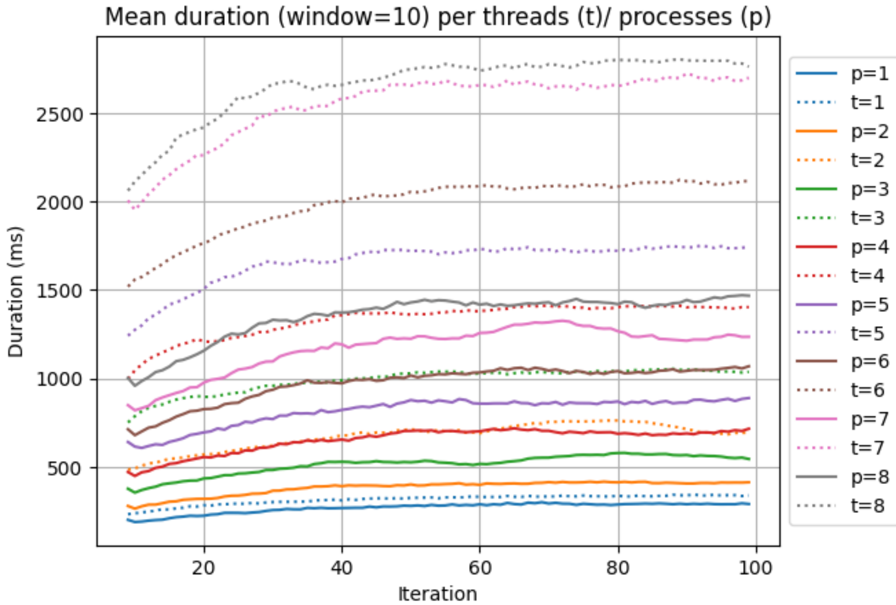
Correctness

We tested both threads and processes by using `parallelism=1` and running the parallel solver along a standard solver. After each iteration we compared the state of both and asserted they matched.

We ran this test for 10 iterations.

Performance

Duration

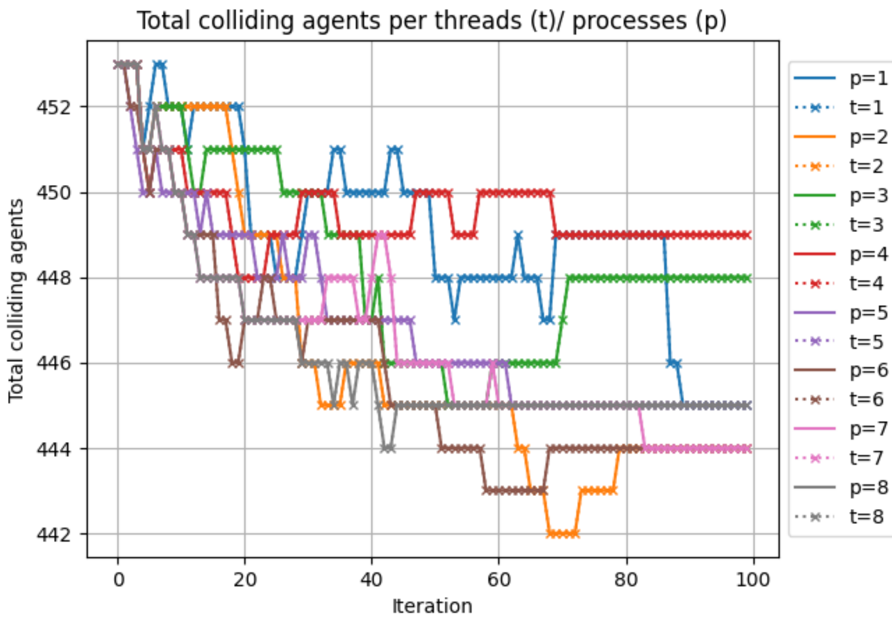


Durations per iteration are noisy, so we plot the rolling mean with a window of size 10.

Each color is for a `parallelism` amount. A solid line uses processes while dotted lines use threads.

Collisions

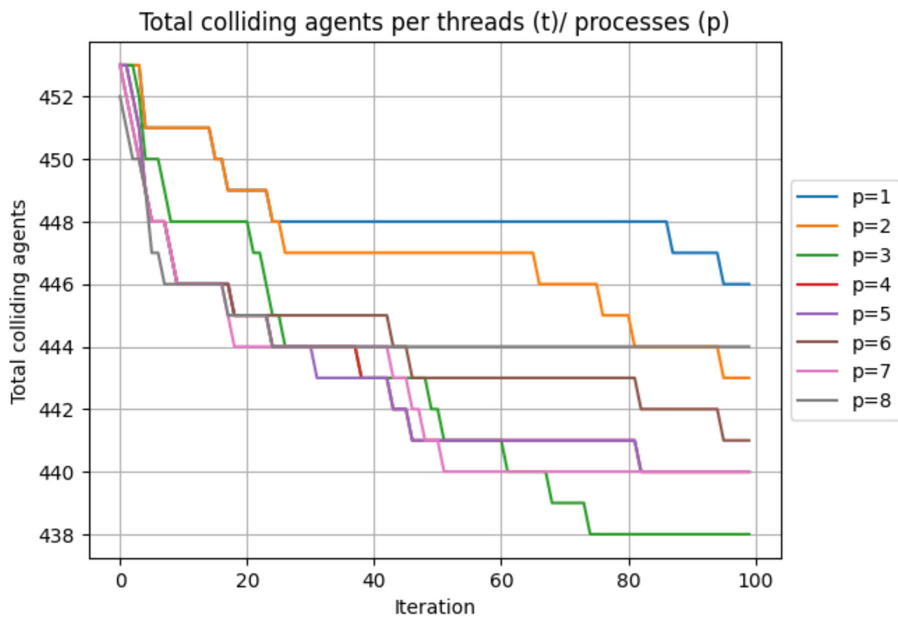
We can use `parallelism=1` as a proxy for no parallelism, since their results are identical.



Y axis is the total count of agents that have at least one collision.

Solid line is for processes, dotted x is for threads. Both are identical showing implementation for both is correct.

Seeing that the unique agent collisions can go up, we changed the `num_collisions` method to calculate this value, instead of the collisions matrix sum, we get lower colliding agents:



Change in code:

```
def num_collisions_in_robots(self, num_robots = 90):
    return self.collisions_matrix.sum() // 2
    return np.sum(np.sum(self.collisions_matrix, axis=1) > 0).item()
```