Semestral Project Report

Parallel programming for combination of OpenMP and MPI for multi-computer systems

1. Introduction

The goal of the semestral project is to transform a computational problem into parallel and multi-computer domain to enhance the problem in term of computation speed or/and quality of output. For such purpose Particle Filter algorithm has been taken.

Particle Filter (PF), of Sequential Monte Carlo (SMC) methods are a set of Monte Carlo algorithms used to solve filtering problems arising in signal processing and Bayesian statistical inference (source: wikipedia.org). The PF has been taken to solve a problem of robot localization with a given map. In addition, odometry and laser scans (LIDAR) data are also given and considered to include measurement error.

Principle of the algorithm is based on initial spawning of random points, each of them represents possible location and orientation of robot. On a move (odometry update), algorithm sequentially propagates the movement to every point (in other words, point is moved in accordance with odometry update, but some noise is also added) and weights them in terms of correspondence to the given environment (by comparing received laser scan with expected in given location). In case point is no more feasible after position update due to movement, it got rejected and new random one spawned instead. Algorithm also adds some fixed number of random points on each step. After particle location and weight update, algorithm randomly picks points with a roulette principle – total sum of weights of all points represent maximum random value, R_{max}, to generate - algorithm generates random values in range (0, R_{max}), so point with a greater weight could be selected more probable than point with a lower weight.

As a result, we can observe particles distributed on map. Most probable particles create a cloud of points around it – example is provided on Fig.1.

Fig.1. Example of Particle Filter output, where white area represents free area, black – occupied area (walls), grey – unknown area. Red points are points generated by PF (actually, not all of them are red, but a gradient from red to yellow based on weight, where most red points have the lowest weight), green points – odometry update, blue – laser scans.

2. Implementation

Source code is located in Github repository and available by following link: https://github.com/Misha91/PAP. The code is not public until the end of the semester.

2.1 Initial profiling

To estimate focus area of the speed up, it's recommended to start with initial profiling. The algorithm consists of three functions – roulette sampler (to pick random particles for next step), move particle (location update based on odometry) and weight update (weighting based on correspondence to the environment). Initial profiling (Fig.2) revealed that main focus should be dealt to the weight update function.

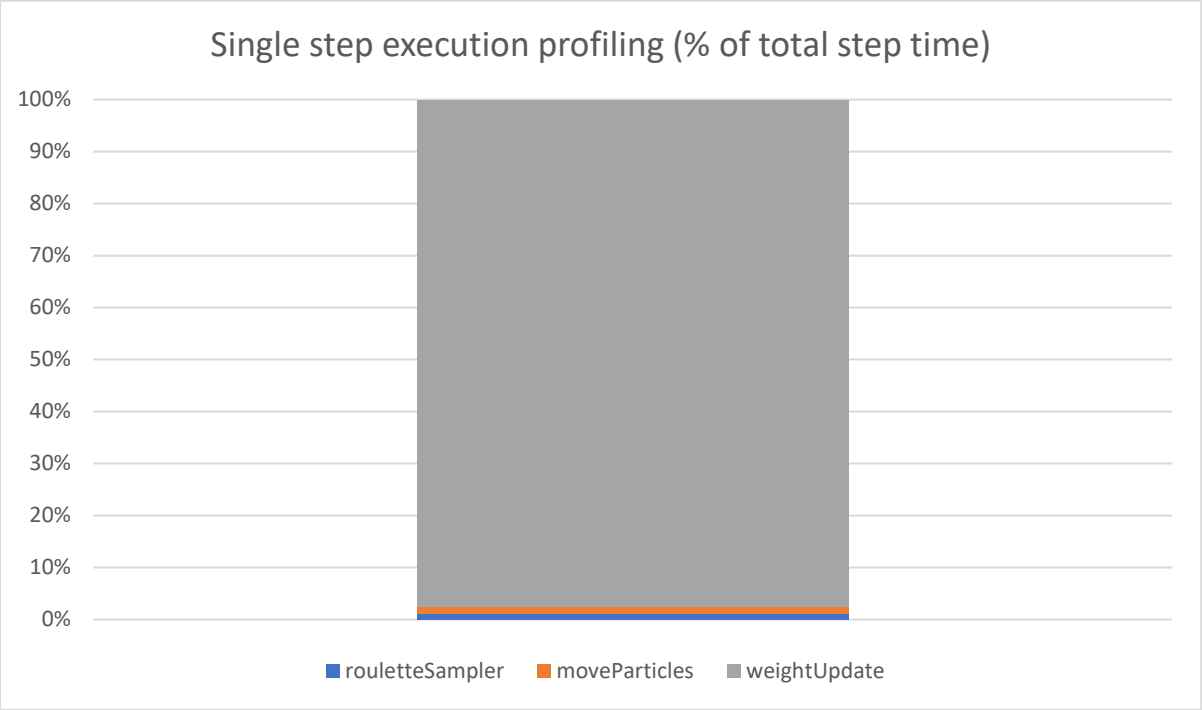

Fig.2 Initial profiling

2.2 OpenMP

The weight update function consists with preparation of laser scan data (we sparse data so every 10th beam is taken, it's enough for weighting purpose and speeds up computation). Then it performs weighting for each point in a for-loop – the area for the improvement. In this area for loop takes initial iterator to the beginning of particles std::vector and iterates it until end is reached.

To parallel the for-loop section, we should use following declaration:

```
#pragma omp parallel for private(prob_beam, z) firstprivate(z_star, simul) shared(max_weight) schedule(auto)
```

We declare omp parallel for, so OpenMP will parallel the for-loop for us. Each thread is required to have a private variable prob_beam and z to perform the cycle, and it will speed up if each thread will have a copy of z_star and simul. We need to share max_weight variable which keeps a point with a greatest weight (it will be needed at OpenMPI section).

Approximately the same approach has been also used for other for loops. Results of the improvement is shown on Fig.3.

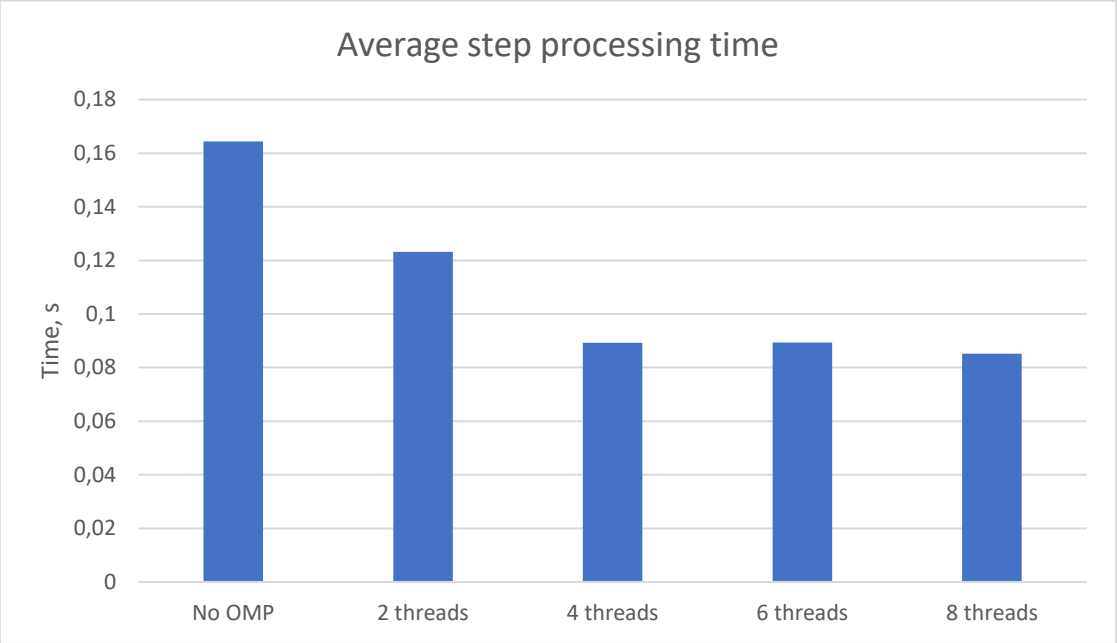

Fig.3. OpenMP speed up performance – execution time based on number of parallel threads.

2.3 OpenMPI

Since FP is a probabilistic method, it's not guaranteed that correct location will be found fast and it will last for long (i.e. algorithm will not lose itself on map). To achieve good results, we need to choose sufficient number of points to track and require both – new points randomly spawned as well as points tracked from the previous step.

With a greater number of points, we can achieve better performance, however there is a threshold, after which is robust enough for given data and doesn't lost itself. Initial localization is a stochastic process, so there is no way to guarantee immediate localization, however with larger number of random points we can expect that it will happen sooner in average.

Another constraint is computational power. Since the algorithm is supposed to be used for robot localization, we cannot expect that robot would bare high-powered computer on itself. In opposite, we'd like to equip a robot with a light-weighted embedded processor and move computation on stationary PC. However, if we have a several robots, OpenMPI could be the thing that will use all the available computational power to solve high-performance computational task.

In this work, we've used OpenMPI to create 4 separate nodes which send points with the greatest weight to a node with rank 0. The node takes those points into its particle list and keep them on next steps. We used it calculate average number of steps needed for initial localization, average number of location lost during the test and ration of localized and localized time. The results are present in Table 1.

Average number of steps for initial localization	
No OpenMPI	OpenMPI (4 nodes)
137,3	59,9
Average number of localization lost	
No OpenMPI	OpenMPI (4 nodes)
1,2	0
Average % of the time robot localized correctly	
No OpenMPI	OpenMPI (4 nodes)
80,26	98,48

Table 1. OpenMPI measurement results

Conclusions

In this work we learnt the principle of computation parallelization and distribution. We got familiar with OpenMP and OpenMPI API and used it to improve performance of Particle Filter algorithm. We found how we can improve localization result and hence algorithm quality by utilizing all available cores as well as available machines.

Mikhail Ivanov

Appendix

Specification of test machine:

Packard Bell EasyNote TS 11 HR

Processor Intel® Core™ i7-2630QM, 4 cores, 2.00 GHz, 6 MB cache

8 GGB RAM DDR3

NVIDIA GeForce GT 730M

Raw data:

Test #	No OpenMP	OpenMP			
		2x	4x	6x	8x
1	0,156296	0,128098	0,091234	0,101696	0,091086
2	0,157876	0,115871	0,095958	0,088327	0,075079
3	0,179772	0,120873	0,075622	0,078485	0,090954
4	0,154258	0,119615	0,091016	0,078861	0,089916
5	0,173862	0,131465	0,092347	0,099469	0,079092
Avg	**0,164413**	**0,123184**	**0,089235**	**0,089368**	**0,085225**

Step time (in seconds)

Initial localization steps	
No MPI	MPI x4
7	21
7	64
206	28
100	75
39	57
90	64
573	95
240	85
80	80
31	30
137,3	59,9

Localization steps