Adaptation of Monte Carlo Localization Algorithm for Parallel Execution in an MPSoC

Aluno: Alzemiro Henrique Lucas da Silva
Professor: Alexandre de Morais Amory

Introduction

In this work we show the implementation of a particle filter used in Monte Carlo localization (MCL) algorithm adapted to be executed in an MPSoC platform called URSA [1]. The application is divided in a ROS node, which runs in a host CPU, and single kernel that run in each CPU of the MPSoC. This implementation resulted in a scalable application, able to run in any amount of CPUs according to the target MPSoC size.

Algorithm

Monte Carlo localization (MCL) is the most popular localization algorithm which represents the belief $bel(x_t)$ by particles. MCL is applicable to both local and global localization problems. The Algorithm below shows the basic operations performed by MCL to apply the particle filtering.

1: **Algorithm MCL**$(\mathcal{X}_{t-1}, u_t, z_t, m)$**:**
2: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
3: for $m = 1$ to M do
4: $x_t^{[m]} = \mathbf{sample\_motion\_model}(u_t, x_{t-1}^{[m]})$
5: $w_t^{[m]} = \mathbf{measurement\_model}(z_t, x_t^{[m]}, m)$
6: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
7: endfor
8: for $m = 1$ to M do
9: draw i with probability $\propto w_t^{[i]}$
10: add $x_t^{[i]}$ to $\mathcal{X}_t$
11: endfor
12: return $\mathcal{X}_t$

Algorithm 1 Monte Carlo Localization based on particle filters [2].

The algorithm receives a set of particles from the previous sample X_{t-1}, a robot control command u_t and a measurement data z_t. At the end of the execution, the algorithm provides a new set of particles for time t, which is X_t.

At the beginning of the execution, two empty sets of particles are created, $\overline{X}_t$ and X_t. Then, for every particle in the set X_{t-1} the motion model is applied and a new particle $x_t^{[m]}$ is generated based in the previous particle position $x_{t-1}^{[m]}$ and the command u_t based in the motion model. In our case, the ROS node provides an odometry information based in its internal motion model, so that our algorithm execute the step shown in line 4 by applying the odometry position difference informed by ROS between the time $t-1$ and time t.

The second step, shown in line 5, is the computation of the particle belief itself w_t based on its new position and the measurement data z_t. In our implementation, this is the step that is calculated in parallel by the MPSoC. Each kernel executing in a core of the system receives all the distances measured by a sensor, z_t, and a particle position $x_t^{[m]}$, to compute the weight $w_t^{[m]}$ of the received particle.

The last step shown in lines 8 through 11 is the application of the filter itself. The basic algorithm show that new particles are “drawn” according to the probabilities calculated in the previous step, and this particles are added to the set X_t which is the set returned by the algorithm. If this algorithm is followed exactly as stated in Algorithm 1, the heavier particles will be replicated several times. To maintain the probabilistic approach of MCL algorithm, the particles are chosen according to its probability and then replicated in the new set of particles with a random distribution noise.

Implementation

This section shows the implementation of the MCL algorithm, described in the previous section, in an MPSoC platform capable of running multiple tasks in parallel. The implementation is divided in a ROS node, which runs on a host CPU, and a kernel replicated in all cores of the MPSoC that are able to compute a single particle weight for each received request.

This implementation was tested and verified using a simulated platform composed of 15 processing cores and a network interface responsible for interacting with the ROS node running in the host CPU via UDP sockets. This platform is organized in a 4x4 square shape, where all processing elements are connected through a Network-on-Chip (NoC). The processing element at position 0 is the network interface, responsible for receiving requests from the ROS node through UDP packets and distribute this requests to the processors running the particle evaluation kernels. Figure 1 shows the basic organization of the platform and how the application runs on it.

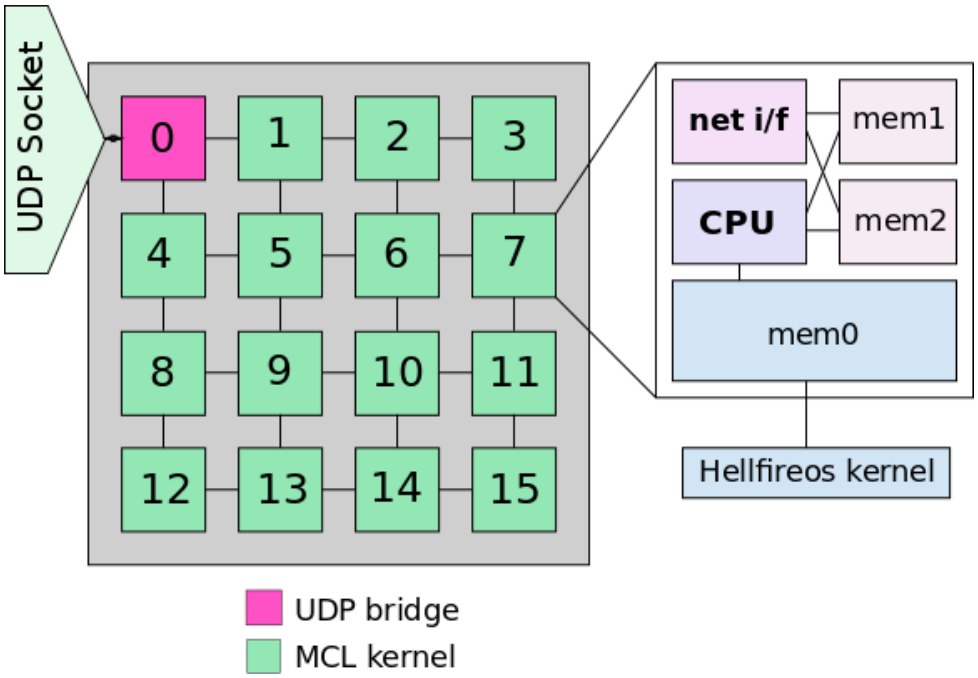

Figure 1 4x4 platform running MCL application.

In this evaluation we used Gazebo simulator to simulate a simple differential drive robot that includes Hokuyo URG-04LX-UG01 laser sensor [3]. The Gazebo simulator publishes simulated sensor and odometry data into ROS topics to be used by external ROS nodes. The ROS node, running at the host CPU, was written to interact with the Gazebo simulation, control the main functions of the MCL algorithm and generate requests to the kernels running on MPSoC to evaluate the particles. Figure 2 illustrates this configuration.

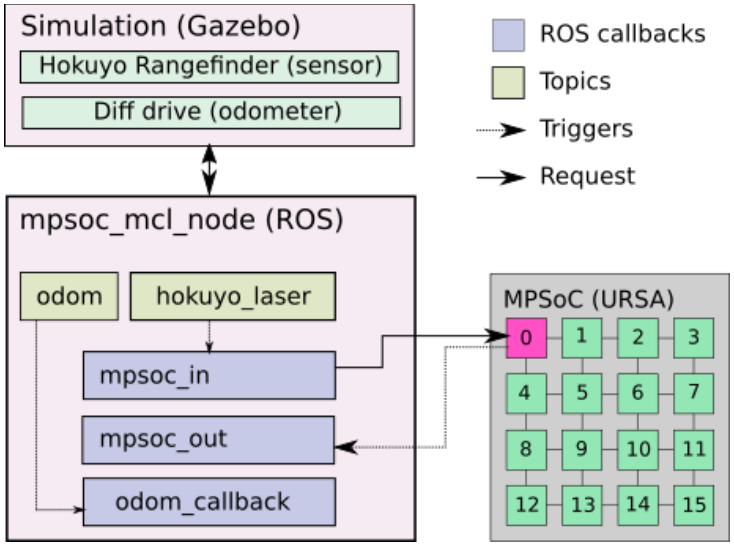

Figure 2 MCL ROS node setup and interaction with MPSoC platform.

The interaction between the MCL ROS node and the kernels running on the MPSoC are done using 4 types of packets:

- DISTANCE_VECTOR: contains 32 sensor distances encoded as 16-bit integers and a particle to be evaluated. The particle values are x, y and θ and are also coded as 16-bit integers. Sensor distances and x, y values are set to have centimeter precision, and θ is a radian value multiplied by 100.
- MAP_REQUEST: contains map coordinates calculated by MCL kernels that runs on the MPSoC for each particle. Due to a limitation of the packet size, each packet contains 16 x, y coordinates, so that two packets are necessary to request all the map positions for each particle. x, y values are coded in 16-bit integer with centimeter precision.
- MAP_VALUES: contains the response of the ROS node with the map values from each x, y coordinate requested by the MCL kernels. Each response contains 16 8-bit values that represents the possible distance between the coordinate requested to the closest object in the map.
- WEIGHT_REPONSE: contains the weight of a single particle computed by a MCL slave kernel. This weight is a probability encoded in a 32-bit fixed point fractional number.

All packets also includes a 2-byte header that carries the type of packet being sent.

It is important to notice that due to a limitation of the platform, the map could not be stored on the MPSoC, so the map was read by the ROS node and the request response protocol for map data was implemented to enable the prototyping of the MCL algorithm in the platform.

Hokuyo provides 640 ranges with an angle span of 240º, but usually the MCL algorithm works well with fewer measures. As the platform provides packets with up to 128 bytes and some bytes are used as header, we decided to use 32 measures of 2 bytes each and a particle data to compose the particle evaluation request packet. Figure 3 shows a sequence diagram that shows how the algorithm is distributed, and the types of packets employed in the communications.

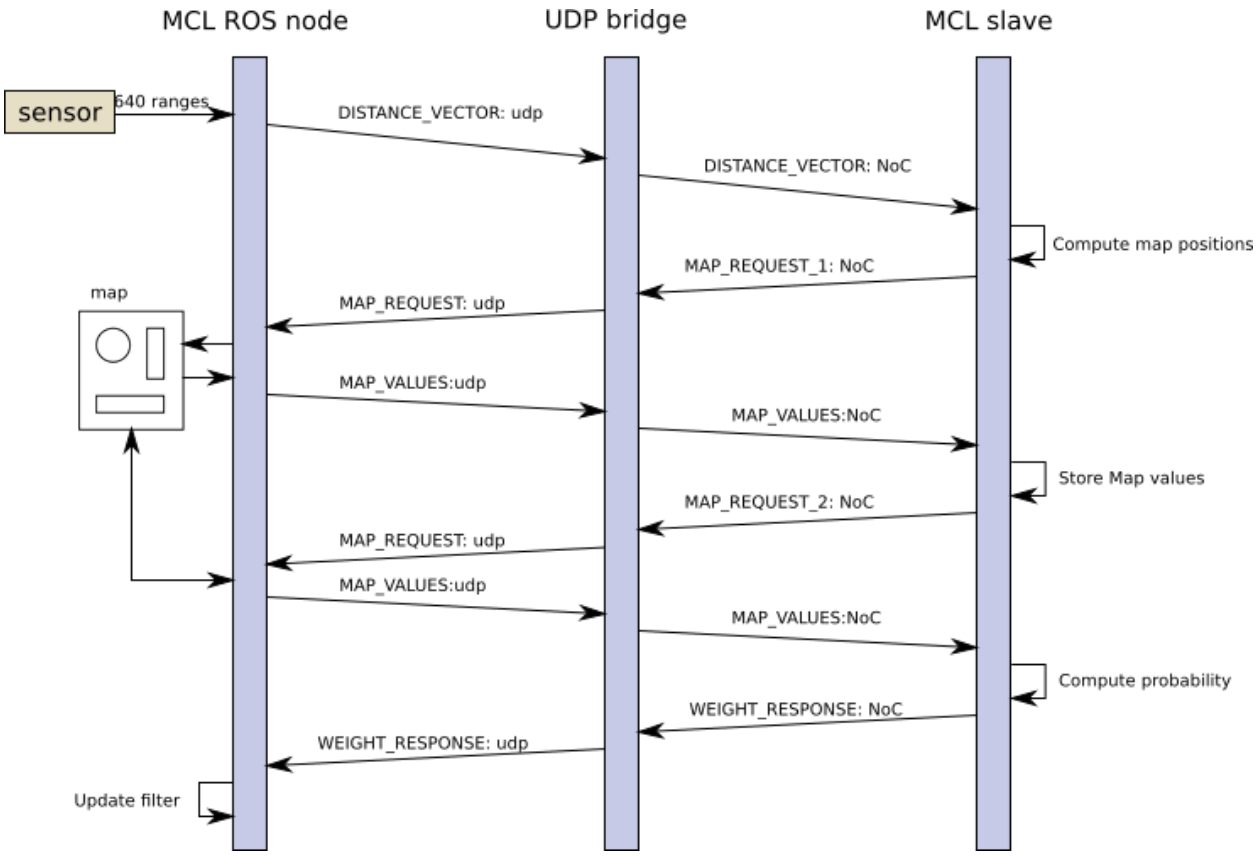

Figure 3 Sequence diagram of the distributed MCL algorithm.

The MCL slave algorithm works by receiving requests composed by a particle position (x, y) and orientation (θ) and 32 ranges from the Hokuyo sensor to compute the resulting map positions and verify the map correlation of the received particle. The computation of the map position is done by applying trigonometric functions based on the sensor reading angle and the particle orientation, multiplying by the received distance. All the sensor distances are related with an angle, which are based on the starting angle and the angle increment for each measure. In this work, we used readings spaced by 20 measures, reducing the total of 640 measures to 32 measures with a known angle as well. To improve the precision of the computation, the 32 known angles for each distance was stored in a table.

The simulation performance of the platform was not enough to build a functional filter, since a good particle filter requires a large initial number of particles to work well. According to [2] 1000 particles would be a good value to start, and by simulating the platform we could compute steadily only 15 particles. Thus, to validate our implementation we draw, in the host ROS node, the coordinates found by the algorithm for a particle to prove that with a larger number of particles the filter could converge. Figure 4 shows the output image for an execution of the application.

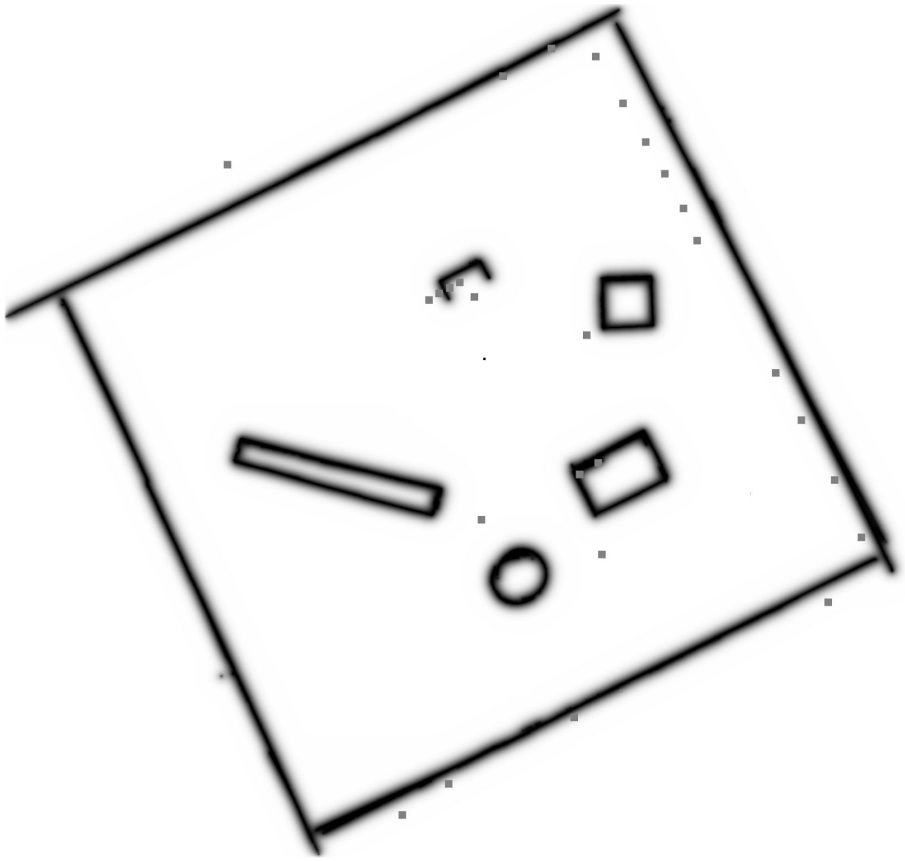

Figure 4 output image of a distance vector computed by a MCL slave.

Conclusions

In this work we show an implementation of the MCL algorithm to run in an MPSoC platform. The algorithm was adapted to be distributed in such way that a host CPU can control the flow of the particle generation and updates, and offload the parallelizable functions to the MPSoC, such as the weight computation. After all weights are computed in parallel by the MPSoC the main node can update particle values and run the filter.

Despite the challenges faced related to the performance of the platform simulation, we show that the adopted method is scalable and functional. Therefore, using the strategy of running a single MCL slave on each core of the MPSoC can increase the performance of the particle evaluation problem and enable more complex algorithms, such as raycasting to verify the the presence of objects between the measure and the robot itself.

References

[1] Anderson R. P. Domingues. 2018. andersondomingues/URSA : An environment or simulating multiprocessed platforms. https://github.com/andersondomingues/ ursa/

[2] S. Thrun, W. Burgard, D. Fox, “Grid and Monte Carlo Localization,” in *Probabilistic Robotics* . Cambridge, Mass., United States: MIT Press Ltd, 2005, pp. 187-219.

[3] LTD. Hokuyo Automatic CO. 2018. Scanning Rangefinder Distance Data Output/URG-04LX-UG01 Product Detail. www.hokuyo-aut.jp/search/single.php?serial=166