Multi-thread impact on the performance of Monte Carlo based algorithms for self-localization of robots using RGBD sensors

Francisco Martín, Vicente Matellán and Francisco J. Lera

***Abstract*—Using information from RGBD sensors requires huge amount of processing. To use these sensors improves the robustness of algorithms for object perception, self-localization and, in general, all the capabilities to be performed by a robot to improve its autonomy. In most cases, these algorithms are not computationally feasible using single-thread implementations. This paper describes two multi thread strategies proposed for self localize a mobile robot in a known environment using information from a RGBD sensor. The experiments will show the benefits obtained when different numbers of threads are compared, using different approaches: a pool of threads and creation/destruction scheme. The work has been carried out on a Kobuki mobile robot in the environment of the RoCKiN competition, similar to RoboCup@home.**

***Index Terms*—localization, 3D maps, RGBD sensors, octree, Multi-threading**

I. Introduction

We are interested in developing software for robots that help people in domestic environments. This software consists of many processes that must be running at once on board an autonomous robot. To carry out household tasks, a robot must perceive objects and people, and be able to interact properly with both. In addition, robots navigate along their environment to perform these tasks, so it is important to have a reliable navigation.

In recent years, 3D sensors are becoming very popular as standard equipment in mobile robots. Nevertheless, in most cases they don't take advance of this information. Maps usually describe 2D information, and created either directly from lasers 2D or transforming their RGBD information to 2D distances. In many works, the software used is based on this idea [1]. While the laser is a very precise sensor, we think they are missing the 3D capabilities of RGBD sensors, which could be very beneficial in environments with symmetries in 2D, but with a rich 3D structure and color. Although the use of RGBD data can improve the capabilities of the robot, it presents a performance problem. The amount of information from this sensor is twice that of a normal image, since each pixel also includes spatial information. The processing of this cloud of points is not easy because usually requires spatial transformations and distance calculations.

Francisco Martín is with University of Rey Juan Carlos.

E-mail: francisco.rico@urjc.es, vicente.matellan@unileon.es, fjrodl@unileon.es

Vicente Matellán and Francisco J. Lera is with University of León, León.

To make a robotic system that requires so much computing resources feasible, it is necessary to pay attention to how the processes that make up the system are executed. Usually, each of these processes usually correspond to a kernel process, delegating the scheduling to the operating system or libraries. Sometimes a single process performs very heavy computation that can not be addressed as a pipeline. Processing point clouds RGBD sensor is an example of this. These sensors are becoming increasingly popular because they provide useful information for reconstruction of objects, mapping and, in general, any process that benefits from having a 3D structure with color. The processing of these clouds of points is computationally expensive. PCL library [2] has become the standard way to handle these point clouds. In addition to its integration with ROS, it offers a lot of tools for segmentation, recognition, feature extraction and correlation points, among others. Many of the functions have a GPU version, which speeds up the processing of these clouds. On other occasions, functions as point clouds searches have not a GPU version available, being a real bottleneck in many applications. In addition, GPU programming requires a low-level programming in which most of the available libraries for scientific computing are not available.

Using multiple threads is a strategy to improve the efficiency of a process with large computing requirements. The optimal number of threads depends on the number of cores of the processor that is running the software. The ideal is to distribute the work in a number of threads similar to the number of cores. Current conventional computers have multiple cores available, and they are not always completely exploited. Most of the implementations use a single thread approach for all the computation.

To validate our research in this field, we participate in robotic competitions that simulate a home environment: RoCKiN [4] and RoboCup@home [3]. The competitions have proved to be a good tool to advance robotics. They present a common scenario where different groups can evaluate and compare their research using standardized tests which can measure the performance of each proposal. In this competition the robot must develop a series of activities to help a person with a degree of dependence in their home environment. These missions include to receive and recognize visitors, to find objects around the house or taking a glass of water from the kitchen to the bedroom. It is also important that the robot share its environment with humans safely.

In this paper we present two multi thread strategies to perform the self localization based on Monte Carlo algorithm [5] using RGBD information intensively. We will demonstrate how a multi-thread approach can make this approach feasible in real time. In order to distribute computing in multiple threads, we have developed two strategies: a pool of threads and threads that are created and destroyed. We will test both strategies with different numbers of threads, from 1 to twice the cores available. The direct consequence is an improvement in the response time of the self localization algorithm, and thus the execution frequency. The higher this frequency, the greater the speed at which a robot can move safely, thereby improving the time in which a robot can reach from one position to another in the environment.

The remainder of this paper is structured as follows. After discussing related work in the following section, we will briefly present the self localization algorithm to be improved in section III. The main contribution of this paper, the multi thread approach will be presented in section IV. In Section V we then will present experimental results of the multi thread comparison. Finally, in section VI we will present the conclusions and future works.

II. Related work

The organization of the execution of software is critical in robot programming. In a system that requires real-time characteristics, where there are deadlines for performing operations. If these deadlines are not met, the result can not be valid, or even disastrous for operation of the robot. Developing robotic systems with such characteristics have led to several works [6][7] focused on systems and methodologies whose emphasis is real time. In many cases, the real-time conditions are applied to the access to sensors and actuators [8][9], while these conditions are relaxed for rest of the software.

In other cases, soft real-time conditions are enough. These approaches uses a graceful degradation, as in the case of Bica [10], where one thread performs all the computation of the robot. Each module executes at its own frequency, forming an activation tree representing its execution dependencies. The execution is performed in cascade, ensuring the execution frequency of each one. If a process exceeds its time slice, the system degrades delaying the execution of others, doing their best to recover their frequency. Our proposal is capable of performing a similar degradation, but taking advantage of multi-core processors of current architectures. Our proposal manages to avoid concurrency problems, but at the cost of a less generalizable solution.

ROS [11][1] is currently the standard in robot control systems. A robotic application in ROS is composed by several interconnected nodes. Drivers for accessing the robot sensors and actuators are implemented using nodes, that provide a common interface by standard messages. Each node has a thread for the user and others responsible of managing communications. There are no race conditions between these threads because it is explicitly synchronized when data handlers of these message queues are executed and when the user code is executed. From the point of view of the user, the nodes run in a single thread. The way to balance the computation load is to design the complete process as a pipeline of ROS nodes. The operating system scheduler distributes the execution of the nodes in the available resources.

The processing design presented in this paper would have been implemented in ROS creating several ROS nodes that divide the work. Besides the problems of synchronization between processes, the communication delay makes not feasible this way. Our proposal is running on a node ROS, but creating threads inside it in a very controlled way.

A very effective way to speed up processes that require large amounts of computing is to use the power offered by the Graphic Processor Unit (GPU) [12]. These components contain many computing units that offer perform extensive computations in parallel [13]. This approach is useful when the same operation is applied to large amount of input data. GPU parallelization techniques have been used in expensive processes, like training neural networks [14][15] or Neural Gas Accelerated [16][17]. RGBD data processing requires great computing power, so the GPU parallelization techniques have been used extensively in this field [18]. In [20], a SLAM algorithm that uses information from a sensor RGBD is presented. In this paper, it uses a GPU parallelization to make an algorithm real-time properties. Our approach is different because we speed up the process using multiple threads on the CPU, instead of a GPU. On one hand, we get less performance, but on the other hand, it allows the use of standard libraries.

The problem of self localization of a mobile robot has received great attention from the start of the Mobile Robotics. Self localization methods estimate the most probable position of a robot in its environment using the information from its sensors and a from a map. Kalman Filter [21][22][23][24] is one of the first widely used estimator for nonlinear systems. This method maintains a gaussian state estimation of the robot position, and its associated uncertainty. This method has difficulties to start if the initial position is unknown, or when kidnapping situations occur. There are works that try to overcome these limitations, maintaining a population of extended Kalman filters [25].

Other popular methods are called Markovian. These methods represent the environment in states that may correspond to a regular grid [26] or irregular regions [27][28]. The probability associated to each state is updated using Bayes' Theorem, establishing a priori probability of possible observations in each state. The main advantage of this method is that it is *global*. This means that it can maintain multiple hypotheses about the position of the robot and it can recover from situations of kidnappings or unknown initial positions. The main disadvantage of this method is its high computational cost when high precision is required, or when the environment is very extensive. In [29] this problem is addressed by dynamically varying the number of cells and the size of the grid, but this complicates how the probability of some states are updated when the robot moves.

Currently, the most widely used methods are based on particle filters [5][30][31], also called Monte Carlo method [32]. This method is based on sampling a probability distribution by a set of hypotheses, called particles. Those particles most likely will remain in the population, while the less likely ones

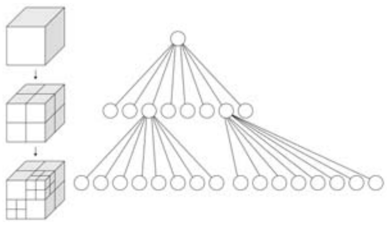
Fig. 1. Map organization in an octree.

will be replaced by others. The robot position is determined by the accumulation of particles with high probability. This method is independent of the size of the environment, and has been used in many applications [33][34]. Furthermore, this method is very flexible and allows many optimizations. In [35], new particles created in each cycle are not generated at random, but taking into account the latest perceptions. It can use a variety of perceptions, achieving a very robust method in highly dynamic environments such as [36] where applied to legged robot in robot soccer matches.

Neither there are enough jobs on using 3D maps for self location. In [37], they use RGBD information, but to find 2D planes, using a 2D map. In [38], the navigation of a robot with legs is made using a 3D map of the environments is made, although the self location information is performed using only a laser distance. Our approach takes full advantage of the position information and color offered by the RGBD sensor and a 3D map of colored dots.

III. Self Localization Algorithm

A. Map

A map contains the knowledge of the environment that the robot uses to self localize. In our work we use RGBD sensors to create the map, so that our map is made up of colored 3D points. A map $\mathcal{M}$ is a set of points with a position and a color in the HSV color space, (x, y, z, h, s, v). This set of points is structured as an octree [39], as shown in Figure 1. It is a tree structure in which each node subdivide the space into eight octants. Using an octree, it is efficient [40] to calculate the points in an area, or the neighbors of a particular point.

In general, building an octree has a complexity $O(Nlog(N))$. Searching on a map has a complexity of $O(log(N))$ in the best case. The search operation $find(\mathcal{M}, p, R)$ returns a set of points $\mathcal{MS} \subseteq \mathcal{M}$ starting from a point p and radius R, in which,

$$find(\mathcal{M}, p, R) \rightarrow \mathcal{MS}, dist(p, p_j) < R, \forall p_j \in \mathcal{MS} \quad (1)$$

Moreover, this set is ordered, so,

$$dist(p, p_n) < dist(p, p_m), \forall p_n, p_m \in \mathcal{MS}, n < m \quad (2)$$

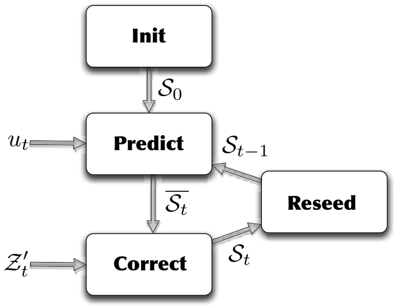

Fig. 2. Monte Carlo Algorithm scheme.

B. RGBD Monte Carlo localization

The state of a robot s can be seen as a displacement (x, y, z) and a rotation (ϕ, θ, ψ), which form an invertible transformation from the origin of coordinates on the map to the robot. In the case of wheeled robot, notation position can be simplified to (x, y, ϕ).

To estimate the robot position, we will use a Monte Carlo algorithm, whose scheme is shown in Figure 2. This algorithm samples the probability distribution $bel(S_T)$ that represents the position of the robot as a set $\mathcal{S}_t$ of hypotheses about the position of the robot S_T, also called particles,

$$\mathcal{S}_t = \{s_t^1, s_t^2, \cdots, s_t^N\} \quad (3)$$

each element of $\mathcal{S}_t$ is a hypothesis S_T associated to an probability so

$$s_t^i \sim p(s_t^i | z_{1:t}, u_{1:t}) \quad (4)$$

The accumulation of particles in a region of the environment indicates the position of the robot.

Initially, S_0 is set depending on the a priori knowledge that we have about the position of the robot. If we start from a known state $s_{initial}$, $s_0^n = s_{initial}, \forall s_0^n \in \mathcal{S}_0$. This problem is usually called *tracking*, because the problem focuses on correcting the errors in the odometry using sensory information. On the other hand, If $s_{initial}$ is unknown, $bel(S)$ is uniformly distributed on the set of possible states of $\mathcal{S}$. This problem is more complex, and is called *global* self location problem, where we have to determine the position of the robot from a situation of complete ignorance. This method is effective to solve both problems.

In the prediction phase, u_t represents the transformation (rotation and translation) of s_{t-1} to s_t measured by the proprioceptive sensors of the robot. This is the expected displacement depending on the control commands generated at each instant t. The application to a state s_{t-1} produces the *prediction* of the state *a priori* s_t, or $\overline{s_t}$.

$$\overline{s_t} = s_{t-1} * u_t \quad (5)$$

In the *prediction* phase, particles in $\mathcal{S}_{t-1}$ are updated using the transformation u_t, applying a noise n_t so

$$n_t \sim N(0, \sigma_u^2) \tag{6}$$

that is a gaussian error that is expected to be u_t, which follows a normal distribution. In wheeled robots, where this approach has been applied, the standard deviation σ_u^2 is low, since the odometry information is very reliable.

In the *correction* phase, we update $\overline{\mathcal{P}_t}$ to $\mathcal{S}_t$ using the perception Z_t. Under normal conditions, Z_t can be composed for nearly 300000 elements. Such amount of information makes computationally not feasible to use this full set to calculate the weight of each particle in $\mathcal{S}_t$. The number of times we calculate $p(\overline{s_t^i}|z_t^j), \forall \overline{s_t^i} \in \overline{\mathcal{S}_t}, \forall z_t^j \in Z_t$ can be 640×480 points $* 200$ particles $= 61440000$. The calculation of $p(\overline{s_t^i}|z_t^j)$ involves comparing each point z_t^j with its neighbors in the map $\mathcal{M}$, which increases the computational requirements of the whole process. In addition, we want run our algorithm several times per second, ideally between [10-20]Hz, to be used by a navigation algorithm.

To make possible the execution of our algorithm to this frequency, we do not use the full perception Z_t, but randomly select a subset $\mathcal{Z}_t'$ of $\mathcal{Z}_t$, so $|\mathcal{Z}_t'| < N$, where $N \in [100-500]$. This range of values facilitates the execution of the algorithm at an acceptable rate, and it is significant enough to update the probability of $\overline{\mathcal{S}_t}$, as will be shown in experimentation section.

For each element of $\overline{\mathcal{P}_t}$ we calculate a weight w_i, which corresponds to a probability given the set of perceptions $\mathcal{Z}_t'$.

$$w_i = \frac{1}{|\overline{\mathcal{S}_t}|} \sum_{i=1}^{|\overline{\mathcal{S}_t}|} \sum_{j=1}^{|\mathcal{Z}_t'|} p(\overline{s_t^i}|z_t^j) \tag{7}$$

$$p(\overline{s_t^i}|z_t^j) = \frac{p(z_t^j|\overline{s_t^i}) * p(\overline{s_t^i})}{p(z_t^j)} \tag{8}$$

Considering that $\overline{s_t^i}$ represents a transformation from the origin of the map, we can calculate the position of z_t^j in the map.

$$l = z_t^j * \overline{s_t^i}^{-1} \tag{9}$$

As we saw in the section III-A, it is possible to obtain a set $\mathcal{MS}$ using the function $find(\mathcal{M}, p, R)$. The probability $p(z_t^j|\overline{s_t^i})$ is calculated from similarity of these two points based on the color difference and in the distance in position.

In the last part of the algorithm, we create a new set $\mathcal{S}_t$ from $\overline{\mathcal{S}_{t-1}}$ after incorporating u_t and $\mathcal{Z}_t$. This phase is known as *resampling*. Figure 3 shows this process. $\overline{\mathcal{S}_t}$ is represented at the top of this figure as an ordered vector, where the color indicates the weight w_i of each particle $\overline{s_t^i}$ in $\overline{\mathcal{S}_t}$. Particles whose w_i is higher are placed at the beginning (in green) and the least likely (in red) are placed at the end.

In our approach we perform resampling in two steps. 50% of the most likely particles remain of $\overline{\mathcal{S}_t}$ to $\mathcal{S}_t$, while the other 50 % are removed. Next, these particles are replaced by others generated from the existing ones in $\mathcal{S}_t$, initialized to $w_i = \frac{1}{|\mathcal{S}_t|}$. In our approach, we use the first 25% of the most likely particles to perform the generation of the new ones.

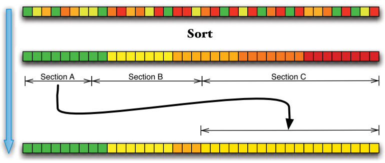

Fig. 3. Resampling of $\mathcal{S}_t$.

Through this process, in each cycle the less likely particles are eliminated, and are replaced by others in positions where it is more likely to be robot. This method is effective to solve the problem of *tracking*, where there is already a set of initial particles is supposed to be close to the actual position of the robot. The new particles will correct errors in u_t with the information from sensory $\mathcal{Z}_t$.

IV. Multi-thread approach

The workflow of the Monte Carlo algorithm is an iterative pipeline, as shown in Figure 2. In the prediction phase, u_t applies to all particles $s_t^i \in \mathcal{S}_{\sqcup - \infty}$, which is not a heavy workload. In contrast, the correction phase needs many more computational resources. For every particle $s_t^i \in \overline{\mathcal{S}_\sqcup}$ a test is performed using the perception points $\mathcal{Z}_t' \subset \mathcal{Z}_t$. This test has several steps:

1) To apply to $z'^i_t \in \mathcal{Z}_t'$ the transformation that s_t^i represents, obtaining l_i.
2) For each point l_i, we make the set $\mathcal{MS}$ using the $find(\mathcal{M}, l_i, R)$ function, where $|\mathcal{MS}| \leq 20$. One reason to use threads instead of GPU is this function $find$, which is the one that consumes more resources. Without a real GPU implementation it becomes in the real bottleneck in this processing.
3) To accumulate the probability calculated from comparing the distance metric and color betweenz'^i_t and every element in $\mathcal{MS}$.

This amount of work is divided into a set of threads $\mathcal{T}$, as shown in Figure 4. It is assigned a subset of $\mathcal{S}_{\sqcup - \infty}$ to each thread, to apply the phases of the Monte Carlo algorithm.

The response time of a process is the time since it starts until the result is obtained. The response time depends on the number of threads $|\mathcal{T}|$ that can run in parallel. On a UNIX system, each thread is mapped to a kernel process, and the operating system scheduler assigns each thread to each processor core. For this reason, when $|\mathcal{T}|$ is greater than the number of processor cores, the response time of the system does not improve.

Creating a POSIX thread on Unix/Linux is relatively inexpensive. As it is a kernel process that shares almost all its memory with the rest of the threads in the same user process, creating a thread consists in reserving space for its stack and its registers. Creating a POSIX thread in C ++ is to create a `std :: thread` object, specifying a function to run and its arguments. The parent thread can synchronize with the new threads through several mechanisms. The simplest is by `join`, which is a function that blocks the calling thread until one of his sons ends.

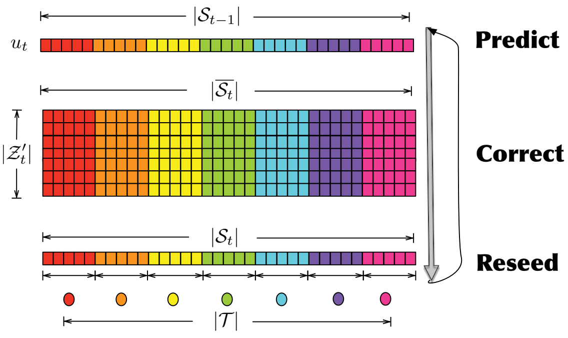

Fig. 4. Workload assigned to each thread, represented as circles, in $\mathcal{T}$.

The following code shows the multi thread approach in which $\mathcal{T}$ threads are created and destroyed in each iteration to process a subset of $\mathcal{S}_{\sqcup-\infty}$. Synchronization follows a `create/join` scheme. For simplicity, we show only the code of the correction phase. Other phases are similar, in terms of number of threads and synchronization scheme.

```
int numParts = 200;
int numThreads = 8;
int numPercepts = 20;
Particles parts[numParts];
Points percept[numPercepts];

void doCorrect(int init, int end)
{
        for(int i=init; i<end; i++)
                update(parts[i], percept);
}

void correct()
{
        thread T[NumThreads];

        for(int i=0; i<NumThreads; i++)
        {
                int start = i*(numParts/numThreads);
                int end = (i+1)*(numParts/numThreads);

                T[i] = thread(doCorrect, start, end);
        }

        for(int i=0; i<NumThreads; i++)
                T[i].join();
}
```

This approach is valid and it works properly. The drawback is that we create and destroy 8 thread per phase in each iteration of the algorithm. If the frequency is 10 Hz, we create 240 threads per second, 14400 threads per minute. Usually, in a Unix system there is a maximum number of threads that can run simultaneously, although this limit does not affect us because it is always 8 in our case. The relevant limit in this case is the maximum number of PIDs that it is configured the system. In long operations, this limit is reached easily.

The most convenient approach in this case is to maintain a pool of threads that are created at the beginning. These threads wait to be request to start processing, and when they finish are suspended until they return to be required. Still, we must be careful because:

- It is no possible to abort a thread from a pool of threads.
- It is nos possible to determine when a thread in a pool has finished.

For this reason, we must be very careful when designing a solution based on this approach. The threads of the pool must be perfectly synchronized to run only at certain times, when the data to be processed are available. In addition, it is necessary that the main thread knows when all the threads have finished. To coordinate this type of scheme we can use many mechanisms: mutexes, locks, barriers, conditions, etc. In our implementation we used semaphores. This synchronization mechanism is very simple:

- The semaphore S is created with an initial value of N.
- If a thread calls the `wait()` method of S, N is decremented in 1. If $N < 0$, the calling thread suspends in S.
- If a thread calls the `post()` method of S, N is incremented in 1. If $N > 0$, N blocked threads in S are activated.

```
int numParts = 200;
int numThreads = 8;
int numPercepts = 20;
Particles parts[numParts];
Points percept[numPercepts];
thread T[NumThreads];

void initThreads()
{
        for(int i=0; i<num_threads_;i++)
                start_sems = semaphore(0);

        end_sem=semaphore(0);

        for(int i=0; i<NumThreads; i++)
        {
                int start = i*(numParts/numThreads);
                int end = (i+1)*(numParts/numThreads);

                T[i] = thread(doCorrect, start, end);
        }
}
void doCorrect(int init, int end, int th_id)
{
        while(true)
        {
                start_sems[th_id]->wait();
                for(int i=init; i<end; i++)
                        updateProbs(parts[i], percept);
                end_sem->post();
        }
}

void correct()
{
        for(int i=0; i<num_threads_; i++)
                start_sems[i]->post();
        for(int i=0; i<num_threads_;i++)
                end_sem->wait();
}
```

In the previous source code shows that each thread `T[i]` that is created to perform the correct phase has its own semaphore, `start_sems[i]`, initialized to 0. Threads are created after initializing these semaphores. All of them run the `doCorrect()` function. The id argument is used by each thread in this function to identify its own semaphore `start_sem[id]`. All the threads then are blocked in each one's semaphore. The main thread also have its own semaphore, `end_sem`, initialized to 0, that is used to block it while the other threads are processing their data.

When the main thread executes the `correct` function, it wakes the other threads `T[i]` calling the `post( )` of each `start_sems[i]`. The main thread can not leave the function `correct()` until each threads has completed its

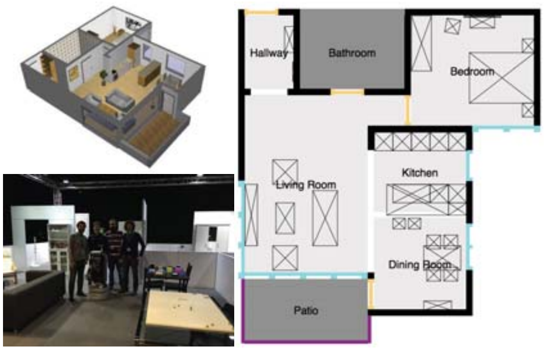

Fig. 5. RoCKiN environment.

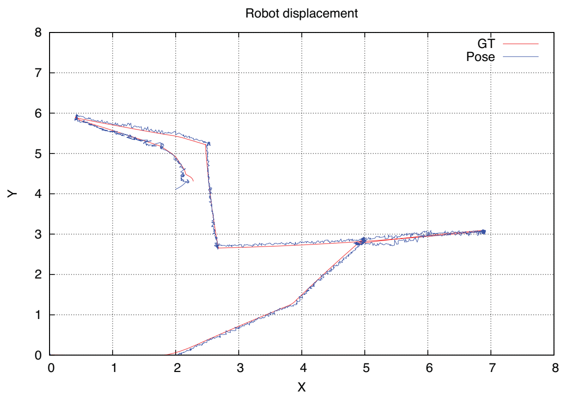

Fig. 6. Route carried out by the robot. Red line is the actual position and the blue one is the position estimated by our algorithm.

work, so call N times the operation `wait()` of `end_sem`. When each thread `T[i]` finishes its work, it calls the `post()` method of `end_sem` to notify to the main thread this finalization. The last thread `T[i]` in finishing makes the main thread wakes up. Then, each thread `T[i]` blocks until next iteration.

V. Experiments

The self location method proposed in this paper has been implemented to execute on board of two different robots: RB-1 and Kobuki. Both robots move by wheels, they have a laser and RGBD (Asus Xtion) sensor. The robot RB-1 has a computer on board Intel i7 with 8 cores and 8Gb of RAM. The Kobuki robot has no onboard computer. In this case, we have equipped with a laptop with similar features to the computer aboard the RB-1.

First we will show the results of an experiment measuring the validity of the localization algorithm. The aim of this paper is not the reliability and robustness of algorithm but the multithread strategies to implement it. Still, we will demonstrate that the implemented algorithm functioning properly. Therefore, we will show an experiment conducted in the environment of the RoCKiN competition, shown in Figure 5. In this experiment, the robot will follow a route from the Hallway of the house to the Dinning Room, and then to the bedroom. In total, the robot will travel 20 meters in not autonomous mode (Figure 6). These accuracy results are independent of the level of multithreading of the self localization algorithm. The results show that this algorithm is robust and accurate.

Once validated the algorithm, we will conduct an experiment designed to determine the improvement obtained using a multithreaded approach. For this goal, we have implemented the algorithm of self location with both the create/join scheme as a pool of threads. Each scheme has been running for 1000 iterations, showing the average response time, shown in Figure 8. In this experiment we have established an amount of 1 to 16 threads in each phase of each iteration of the algorithm of self localization.

In the case of a thread, the average response time per iteration are 113 ms, similar in both experiments. As increasing the number of threads, the difference between the two strategies is

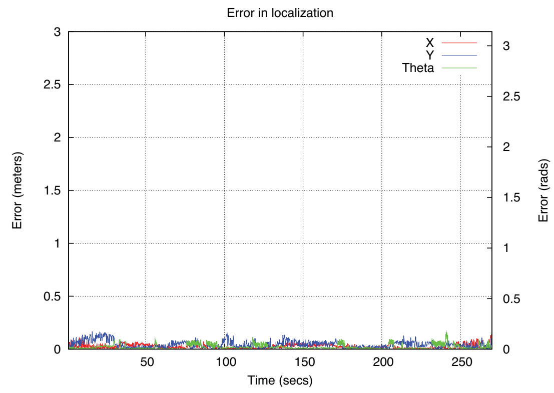

Fig. 7. Error en la localización

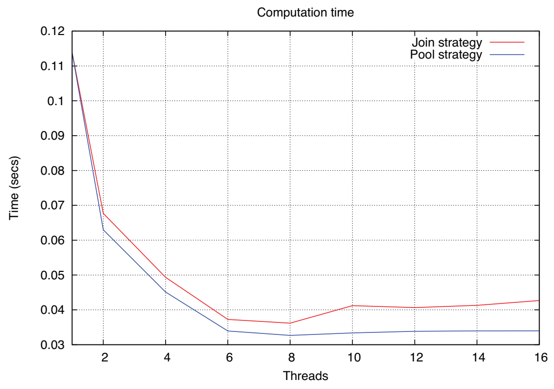

Fig. 8. Average response time both strategies and different number of threads.

increased due to the cost of creation and destruction of threads. The response time increases as we decrease the number of threads. The minimum is around 8 threads, since we are running this experiment on a CPU with 8 cores. Further thread increase does not improve the response time, and it begins to deteriorate in the case of the create/join strategy, due to the cost of the threads creation/destruction. It is important to note that we managed to reduce the response time from 113ms to 33ms. The let us to run the algorithm at 30 Mhz if necessary, instead of the initial 9 Mhz.

VI. Conclusions y and future work

In this paper we have presented a study of the impact of using multi thread mechanisms to address a problem that requires intensive computation, such as a location with RGBD sensors. Naturally, the use of multi threading in multi core CPUs reduces the response time of the algorithms. We have shown how to improve the response time of the algorithm when using as many threads as cores, becoming counterproductive if this amount is higher.

In addition, we measured the overhead produces by a create/join thread scheme rather than the one based on a thread pool scheme. In addition, we have shown how to synchronize the threads of this pool of threads using semaphores. Using this scheme, threads are waiting a signal to start processing. At finalization, threads signal for continuing the execution of the main thread.

We have described an algorithm based on self location Monte Carlo algorithm, using an RGBD sensor. The use of colored dots in the mapped space, and their use in novel self localization. The disadvantage is that this algorithm requires many computational resources. This paper presents the benefits of addressing this problem throughout a multi threading approach. This work has been tested in the last competition ROCKIN 2015 in Lisbon, shown that it works properly and it is able to keep the robot located with an acceptable response time.

One of the future works is to apply GPU parallelization techniques and compare the results with the results obtained in this article. Still, we continue to believe that GPU programming requires a low-level programming and limits the use of libraries have not a GPU version available.

Acknowledgment

The authors would like to thank the Spanish Ministry of Economy and Competitiveness for the partial support to this work under grant DPI2013-40534-R. The authors would like to thank Francisco J. Ballesteros and Enrique Soriano, the Ubiquitous Systems Laboratory, for their valuable comments on the implementation of synchronization thread pool.

References

[1] Marder-Eppstein E., Berger E., Foote T., and Gerkey, B. and Konolige K., *The Office Marathon: Robust Navigation in an Indoor Office Environment*. International Conference on Robotics and Automation. 2010.

[2] Rusu R.B. and Cousins S., *3D is here: Point Cloud Library (PCL)*, IEEE International Conference on Robotics and Automation (ICRA). 2011.

[3] Holz D., Ruiz-del-Solar J., Sugiura K., and Wachsmuth, S.,*On RoboCup@Home - past, present and future of a scientific competition for service robots*, Lecture Notes in Computer Science, Vol. 8992, pp. 686–697, 2014.

[4] P. U. Lima, D. Nardi, L. Iocchi, G. Kraetzschmar, and M. Matteucci, *RoCKIn@Home: Benchmarking Domestic Robots Through Competitions*, In Robotics Automation Magazine, IEEE, vol. 21, no. 2, pp. 8-12, 2014.

[5] Fox, D., Burgard, W., Dellaert, W., Thrun, S., *Robust Monte Carlo localization for mobile robots*, Artificial Intelligence. Vol. 128, pp. 99–141. 2001.

[6] Brega R., Tomatis, R. and Arrast, K., *The Need for Autonomy and Real-Time in Mobile Robotics: A Case Study of XO/2 and Pygmalion*, Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000), Takamatsu, Japan, Oct. 30 - Nov. 5, 2000. Journal in Engineering Applications of Artificial Intelligence. Vol. 17, Issue 5, pp. 469–483. August 2004.

[7] Ce Li, Takahiro Watanabe, Zhenyu Wu, Hang Li and Yijie Huangfu. *The Real-Time and Embedded Soccer Robot Control System*, Robot Soccer, Vladan Papi (Ed.), InTech, DOI: 10.5772/7352. 2010.

[8] Gu, J.S. and de Silva, C.W., *Development and implementation of a real-time open-architecture control system for industrial robot systems*,

[9] Simpkins, A., *Real-Time Control in Robotic Systems*, Applications, Control and Programming, Dr. Ashish Dutta (Ed.), InTech, DOI: 10.5772/27883. 2012.

[10] Martín, F., Agüero, C, Plaza, J.M., *A Simple, Efficient, and Scalable Behavior-based Architecture for Robotic Applications*, ROBOT'2015 Second Iberian Robotics Conference.

[11] Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T.., Leibs, J., Wheeler, R., and Ng, Andrew Y., *ROS: an open-source Robot Operating System*. ICRA Workshop on Open Source Software. 2009.

[12] Satish, N., Harris, M., Garland, M., *Designing Efficient Sorting Algorithms for Manycore Gpus. NVIDIA Corporation*, 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 2009.

[13] Kirk, D., Hwu, W., *Programming Massively Parallel Processors: A Hands-On Approach*, Morgan Kaufmann Ed., 2010.

[14] Parigi, G., Pau, D., Piastra, M., *GPU-based Parallel Implementation of a Growing Self-Organizing Network*, Proceedings of ICINCO 1, pp. 633-643. Jan 2012.

[15] Jang, J., Park, A., Jung, K., *Neural network implementation using CUDA and Open MP*, Proceedings of the International Conference on Digital Image Computing: Techniques and Applications, DICTA 2008, Canberra, ACT, Australia, Dec 2008.

[16] Orts, S., Garcia, J., Serra, J.A., Cazorla, M., *3D Model Reconstruction using Neural Gas Accelerated on GPUs*, Applied Soft Computing, Vol 32, pp. 87-100. July 2015.

[17] Orts, S., Garcia, J., Viejo. D., Cazorla, M., *GPGPU implementation of growing neural gas: Application to 3D scene reconstruction*, Journal of Parallel and Distributed Computing. Oct 2012

[18] Amamra A., Aouf, N., *GPU-based real-time RGBD data filtering*, Journal of Real-Time Image Processing, pp. 1–8, Sept 2014.

[19] Wasza, J., Bauer, S., Hornegger, J., *Real-time Preprocessing for Dense 3-D Range Imaging on the GPU: Defect Interpolation, Bilateral Temporal Averaging and Guided Filtering*, IEEE International Conference on Computer Vision (ICCV), pp 1221–1227. Dec 2011.

[20] Lee, D., *GPU-based real-time RGBD 3D SLAM*, Proceedings of the 9th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI), pp 46–48. 2012.

[21] Rudolph E. Kalman, *A New Approach to Linear Filtering and Prediction Problems*. Transactions of the ASME, Journal of Basic Engineering, Vol. 82, No. Series D, pp. 34–45, (1960).

[22] Lastra, R., Vallejos, P. and Ruiz-del-Solar, J,*Self-Localization and Ball Tracking for the RoboCup 4-Legged League*, Proceeding of the 2nd IEEE Latin American Robotics Symposium LARS 2005, Santiago de Chile, Chile (2005)

[23] Tesli, Luka and A. krjanc, Igor and Klanaear, Gregor, *EKF-Based Localization of a Wheeled Mobile Robot in Structured Environments*, Journal of Intelligent & Robotic Systems, Vol. 62-2, pp. 187–203. 2011.

[24] Hamzah Ahmad and Toru Namerikawa, *Extended Kalman filter-based mobile robot localization with intermittent measurements*, Systems Science & Control Engineering, Vol. 1-1,pp. 113–126, 2013.

[25] Martín F., and Matellán V., and Barrera P., and Cañas, J.M., *Localization of legged robots combining a fuzzy-Markov method and a population of extended Kalman filters*, Robotics and Autonomous Systems, Vol. 55, pp 870–880, 2007.

[26] Dieter Fox, Wolfram Burgard and Sebastian Thrun, *Markov Localization for Mobile Robots in Dynamic Environments*, Journal of Artificial Intelligence Research, Vol. 11, pp. 391–427, 1999.

[27] Sebastian Thrun and Arno Bücken, *Integrating Grid-Based and Topological Maps for Mobile Robot Navigation*, Proceedings of the AAAI

Thirteenth National Conference on Artificial Intelligence, Vol. 2, pp. 944–950. Portland, OG(1996)

[28] Martín F., and Matellán V., and Barrera P., and Cañas, J.M., *Visual Based Localization for a Legged Robot*, Lecture Notes on Computer Science. Vol. LNAI-4020. pp 708–715. (2006).

[29] Martín F., *Visual Localization based on Quadtrees*, ROBOT'2015 Second Iberian Robotics Conference, Volume 418 of the series Advances in Intelligent Systems and Computing pp 599-610. 2015.

[30] Sebastian Thrun, *Particle Filters in Robotics*, Proceedings of the 18th Annual Conference on Uncertainty in Artificial Intelligence (UAI-02), pp. 511-518. San Francisco, CA (2002).

[31] S. Thrun, M. Beetz, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schultea and D. Schulz, *Probabilistic Algorithms and the Interactive Museum Tour-Guide Robot Minerva*, International Journal of Robotics Research, Vol. 19, No. 11. (2000), pp. 972-999.

[32] Dieter Fox, Wolfram Burgard, Frank Dellaert, Sebastian Thrun, *Monte Carlo Localization: Efficient Position Estimation for Mobile Robots*, Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99), pp. 343-349. Orlando, FL. (1999).

[33] R. Conde, A. Ollero and J.A. Cobano, *Method based on a particle filter for UAV trajectory prediction under uncertainties*. 40th International Symposium of Robotics. Barcelona, Spain (2009).

[34] Nak Yong Ko, Tae Gyun Kim and Sung Woo Noh, *Monte Carlo Localization of Underwater Robot Using Internal and External Information*. 2011 IEEE Asia-Pacific Services Computing Conference, APSCC. 2011, pp. 410–415. Jeju, South Korea (2011).

[35] Scott Lenser and Manuela Veloso, *Sensor Resetting Localization for Poorly Modelled Mobile Robots*, Proceedings of ICRA-2000, the International Conference on Robotics and Automation, pp. 1225-1232. San Francisco, CA (2000).

[36] T. Röfer, T. Laue and D. Thomas, *Particle-filter-based self-localization using landmarks and directed lines*, RoboCup 2005: Robot Soccer World Cup IX, Vol. 4020, pp. 608–615, Lecture Notes in Artificial Intelligence. Springer (2006).

[37] Biswas J., and Veloso, M,. *Localization and Navigation of the CoBots Over Long-term Deployments*. The International Journal of Robotics Research, Vol. 32-14. pp 1679–1694, 2013.

[38] Maier, D.*Real-time navigation in 3D environments based on depth camera data*, 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids), pp. 692–697. 2012.

[39] Eberhardt H., Klumpp V.,, Uwe D. Hanebeck, *Density Trees for Efficient Nonlinear State Estimation*, Proceedings of the 13th International Conference on Information Fusion, Edinburgh, United Kingdom, July, 2010.

[40] Hornung A., Wurm K. M., Bennewitz M., Stachniss C., Burgard W., *OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees*, Autonomous Robots, Vol. 34-2, pp. 189–206, 2013.