C++ Multithreading applied to robotics in EKF-SLAM

Daniel Casado Herraez
dcasadoherraez@keio.jp

***Abstract*—With the empirical Moore's law, the number of transistors in a dense integrated circuit doubles each year. However, Dennard's scaling law has been broken, and since 2005 the frequency has began to stagnate. The most common way to deal with this has been parallel computing, where operations are spread over multiple cores. This parallelization is becoming popular in robotics due to their concurrent nature. This report will study from a practical perspective how to apply parallelization for multiple tasks in a C++ robotics context, demonstrating its usefulness in a simplified Extended Kalman Filter SLAM framework.**

Keywords: Robotics, Robot Operating System, Concurrent Programming, C++, Extended Kalman Filter, SLAM.

I. Introduction

C++ has become one of the most popular programming languages to robotics. It provides low real-time performance, Object Oriented Design capabilities and easier to manage low level control than C. Since C++11, multithreading is supported providing a simple interface to the programmer. The topic can be divided into two main areas, threading, and shared resources.

In order to fully understand the scope of the topic, a 2D Extended Kalman Filter SLAM framework has been developed with multiple added parallelizations. The performance will be compared, and the tradeoffs will be studied.

II. Multithreading

The creation of threads in C++ comes from `std::thread` library. Threading provides the programmer with parallelization capabilities in multiple threads. Their creation can be based in four main approaches, functions, functors, lambda functions and asynchronous tasks. They are described at [1], and this report will survey them with some additional comments.

A. Using functions

When using a function in the thread creation, this function will be assigned to that thread taking its function parameters. There are two main cases for creating a thread using functions. Using a function, or using a class method. In both cases, if the argument is to be passed using the reference wrapper `std::ref`.

Using a function does not require any additional considerations.

```
std::thread my_thread =
    std::thread(MyFunction, arg1, arg2,
    ref(arg3), ref(arg4))
```

When using a class method, the reference to the class method must be passed, as well as the instance in which it is being called.

```
std::thread my_class_thread =
    thread(&MyClass::MyFunction,
    class_instance, arg1, ref(arg2));
```

As soon as the thread is created, it will start processing. To wait for a thread to finish, it is necessary to use .join (`my_thread.join()`).

B. Using functors

Similar to the creation of threads using functions, the functors will use instances of a class to perform certain operations instead of calling the same function with different inputs. The function operation will be defined in its `operator()` method.

C. Using lambda functions

An alternative way to initialize threads in a compact way is to use lambda functions. These provide inline functions without an identifier. It has four main components:

```
[capture](parameters) return_type {
    func_body }
```

Capture refers to the output variables and parameters is the arguments to the function (empty if none).

The initialization of the thread is very similar to the previous cases.

```
std::thread([arg1] (int x) {
     // do something with arg1
     return x;})
```

D. Using async

All the previous thread initializaiton methods are expected to run synchronously using the `.join()` method. But C++ also makes possible to run tasks asynchronously such that a receiver is notified after the task is finished.

However, using `std::async` blocks the main thread until the computation is done. So how is this different from ‘std::thread‘?

There are two main points to note. First, `std::thread` does not give access to the function return value, while `std::async` returns a future whose value is obtained via `.get()`. Second, asynchronous calls can be implemented with an asynchronous policy, that can specify how existing threads are assigned. Meanwhile threading will always create a new thread.

III. Mutex and conditional variables

Using multiple threads can be of great advantage for task parallelization. However it doesn't come without drawbacks. Unlike processes, threads have a shared memory component that must be correctly handled by the programmer, otherwise potential deadlocks and race conditions will occur.

The best optimization in this case is to reduce as much as possible the shared data between threads. This might not always be possible, thus mutexes, locks, atomic operations and abstraction levels help the programmer achieve this. This report will survey the most popular ones in this case, mutexes, locks and conditional variables.

A. Mutexes and locks

Defining a mutex means to establish a critical section for a shared variable in which only one program at a time can access. There will be one mutex per critical section, and multiple critical sections can be established within a program.

It can be managed in C++ using direct locks, lock guards, shared locks, unique locks, multiple locks.

1) Direct lock: The direct lock is the simplest form of a lock to access the mutex critical section. While one thread is holding a lock, the other will wait. However when an exception occurs, the mutex stays locked, causing a potential deadlock. In addition, it should be locked and unlocked manually by the programmer.

2) Lock Guard: Lock guard handles the disadvantage from the direct lock by unlocking as soon as it goes out of scope. It follows the RAII programming technique (Resource Acquisition as Initialization), and avoids deadlock if an exception or error occurs.

3) Unique Lock (Lock Guard + Lock/Unlock): The unique lock in C++ combines lock guard RAII with lock/unlock capabilities from the direct lock. It can be used to avoid long wait times where operations on non critical sections are being performed inside a function.

4) Shared Lock: The shared lock allows for multiple reading operations from multiple threads, while having a single writer thread. In this strategy, the writer will hold a unique lock to enter the critical section until the task is finished. Then the readers will hold the shared lock to read the data. Note that the read and write operation cannot occur at the same time.

5) Multiple locks: When using multiple locks, there is a risk of a potential deadlock if not addressed properly. A popular example in this context is the dining philosophers problem. A scoped lock attempts to solve this by locking all mutexes or none of them at the same time, and releases once out of scope `std::scoped_lock(m1, m2)`.

B. Conditional variables

Another sharing mechanism between threads has a producer/consumer pattern, in which one thread produces data and another one consumes it. Therefore there must be some synchronization between them. Shared memory and conditional variables can be used in this case.

1) Shared memory: First, regarding shared memory, a critical section is defined in which data is written and a flag is set. The producer puts data into the shared memory and sets a ready flag. The consumer checks for ready flag, and consumes it when available. This is a useful implementation for cases where data is being continuously read and written, such as the drawer for the EKF SLAM application from the next section in the report. However it performs busy waiting, meaning that the inefficiency is high.

2) Conditional variable: The conditional variable improves the previous shared memory communication by setting a notifier in the critical section. It is based in a two way handshaking between producer and cosumer. This improves efficiency and avoids the busy waiting from the previous approach. The steps are summarized as follows.

1) Create critical section variables (mutex, data, ready, condition variable)
2) Producer: a) Gets lock b) Produces data c) Sets ready flag to true d) Unlocks e) Notifies consumer (`m.notify_one()` or `m.notify_all()`) f) (Optional) waits for response to continue producing
3) Consumer: a) Gets lock b) If blocked, unlocks so that producer can take it (`l.unlock()`) c) If unblocked, locks so that consumer can sample the data (`l.unlock()`) d) Waits until ready notification has been received `m.wait(lock1, []() { return ready;})` e) Gets data coming from the producer f) Unlocks the critical section g) (Optional) notifies producer to continue producing h) Consumes the data.

Note that the wait function has the predicate `ready`, it is essential to avoid spurious wake ups by the OS. Spurious wake ups can cause race conditions by trying to access the same variable from the consumer and the producer.

Moreover, C++ sequential consistency (`unlock()` synchronizes with the next `lock()`) ensures that the notification will be seen by the other thread.

IV. Parallelized EKF-SLAM

After understanding all the practical concepts on C++ parallel programming, it is time to observe how it can be applied to a real scenario. This simulation will perform Simultaneous Localization and Mapping (SLAM) of a robot in a 2D world. The robot measurement is based on the odometry motion model, and a range-bearing sensor to observe the landmarks. An introduction to the problem will be provided. The parallelizable sections will be studied, and the performance will be demonstrated.

The problem formulation is based on Cyrill Stachniss lecture on robot mapping [2] (University of Freiburg) as well as 'Simultaneous localization and mapping with the extended Kalman filter' by Joan Solà [3]. The sensor data was generated by the same professor.

A. EKF-SLAM

The Extended Kalman Filter was one of the first approaches of SLAM by performing a local linearization of a nonlinear system. It couples the current state of the robot only to the previous state and estimates is probability of being at the current state. The algorithm is divided into two main steps, prediction and correction.

The state vector has a $3+2N$ length where N is the number of landmarks. It is defined as

$$x_t = (x, y, \theta, m_{1,x}, m_{1,y}, m_{2,x}, m_{2,y}...)$$

Where $m_i = (m_{i,x}, m_{i,y})$ are the coordinates of the landmarks.

The covariance matrix is $2N$ x $2N$ defined as

$$\Sigma_t = \begin{pmatrix} \Sigma_{xx} & \Sigma_{xm} \\ \Sigma_{mx} & \Sigma_{mm} \end{pmatrix}$$

The input vector (odometry vector) is defined as

$$u_t = (v_t, \delta_1, \delta_2)$$

The measurements come in the form of id, range and bearing $z_i = (id, r_i, \phi_i)$, and they are interpreted as an unordered map in the C++ code.

Note that all the measurements must come already asscociated into the algorithm.

1) Prediction step: In the prediction step, the estimated state is computed based on the motion model (Eq. 1) and an initial uncertainty (covariance) matrix is predicted from previous uncertainty and the motion model jacobian G_t (Eq. 2).

$$x_t^- = x_{t-1} + \begin{pmatrix} v_t cos(\theta + \delta_1) \\ v_t sin(\theta + \delta_1) \\ \delta_1 + \delta_2 \end{pmatrix} \tag{1}$$

$$\Sigma_t^- = G_t \Sigma_{t-1} G_t^T + R_t \tag{2}$$

The jacobian of the motion model corresponds to the top-left component of a $3 + 2N$ identity matrix, and it is defined as

$$G_t^x = \begin{pmatrix} 1 & 0 & -v_t sin(\theta + \delta_1) \\ 0 & 1 & v_t cos(\theta + \delta_1) \\ 0 & 0 & 1 \end{pmatrix}$$

At this point, it is interesting to expand the computation of the covariance matrix and observe it can be decomposed into three parts (Eq. 3).

$$\Sigma_t^- = \begin{pmatrix} G_t^x \Sigma_{xx} (G_t^x)^T & G_t^x \Sigma_{xm} \\ (G_t^x \Sigma_{xm})^T & \Sigma_{mm} \end{pmatrix} \tag{3}$$

This is a possibility of parallelizing computations, as the components do not depend on each other.

2) Correction step: After the first pose and covariance estimate, the correction computes a gain to reduce the error based on the landmark observations. As it will be noted, parallelization can be introduced at this stage to improve the performance.

First, if the landmark has not been initialized yet, its location within the map is computed as the sum of the robot pose estimate plus the relative range sensor measurement (Eq. 4).

$$m_i = x_t^-(x, y) + \begin{pmatrix} r cos(\phi_i + x_t^-(\theta)) \\ v_t sin(\phi_i + x_t^-(\theta)) \end{pmatrix} \tag{4}$$

In order to be able to compare the expected distance from landmark to robot, the squared euclidean distance q is computed as Eq. 5.

$$\begin{aligned} \delta &= m_i - x_t^-(x, y) \\ q &= \delta \delta^T \end{aligned} \tag{5}$$

Once defined, the predicted measurement is given as Eq. 6.

$$z_t^{i-} = \begin{pmatrix} \sqrt{q} \\ atan2(\delta_y, \delta_x) - x^-(\theta) \end{pmatrix} \tag{6}$$

The jacobian of the predicted measurement will have the size $2N$x$2N+3$. In the subcomponent for each landmark, the first three columns refer to the derivative with respect to the robot pose. The other two non-empty columns correspond to the observed landmark index (Eq. 7).

$$H_{low,t}^i = \frac{1}{q} \begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & 0 & ... & \sqrt{q}\delta_x & \sqrt{q}\delta_y & .. & 0 \\ \delta_y & -\delta_x & -q & 0 & ... & -\delta_y & \delta_x & .. & 0 \end{pmatrix} \tag{7}$$

It is important to note in this case, that the computation of each landmark does not interfere with each other. Therefore the matrix H can be constructed by blocks, using batches of landmark detections that can be parallelized for computation.

Performance would benefit from this specially in the case where landmarks and features need to be matched, which is out of the scope of this algorithm, and simulated in the experiments as a small time delay.

The last steps of EKF SLAM are trivial and can be observed in the algorithm pseudocode Algorithm 1.

B. EKF SLAM multithreading

As seen in the previous section, there are two main areas of experimentation for multithreading in EKF SLAM. The first one is in the computation of the covariance matrix prediction Σ^-. The second one is batch-computing the jacobian of the observation for each landmark $H_{low,t}^i$.

Two additional threads were used in order to display the robot movement in a 2D image as well as publishing of the current time. All the writing within threads is handled using unique locks for a data mutex in the class instance.

An example picture of the visualization can be seen in Fig. 1.

Algorithm 1 EKF SLAM Pseudocode

```
    In: $x_{t-1}$, $\Sigma t - 1$, $u_t$, $z_t$
    Out: $x_t$, $\Sigma t$
 1: $x_t^- = x_{t-1} + \begin{pmatrix} v_t cos(\theta + \delta_1) \\ v_t sin(\theta + \delta_1) \\ \delta_1 + \delta_2 \end{pmatrix}$
 2: $\Sigma_t^- = G_t \Sigma_{t-1} G_t^T + R_t$
 3: for all observed features $z_t^i$ do
 4:     if $z_t^i(id) \notin z$ then
 5:         $m_i = x_t^-(x, y) + \begin{pmatrix} r cos(\phi_i + x_t^-(\theta)) \\ v_t sin(\phi_i + x_t^-(\theta)) \end{pmatrix}$
 6:     end if
 7:     $\delta = m_i - x_t^-(x, y)$
 8:     $q = \delta \delta^T$
 9:     $z_t^{i-} = \begin{pmatrix} \sqrt{q} \\ atan2(\delta_y, \delta_x) - x^-(\theta) \end{pmatrix}$
10:     $H_{low,t}^i = \frac{1}{q} \begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & \dots \\ \delta_y & -\delta_x & -q & \dots \end{pmatrix}$
11: end for
12: $K_t = \Sigma_t^- H_t^T (H_t \Sigma_t^- H_t^T + Q_t)^{-1}$
13: $x_t = x_t^- + K_t(z - z^-)$
14: $\Sigma_t = (I - K_t H_t)\Sigma_t$
```

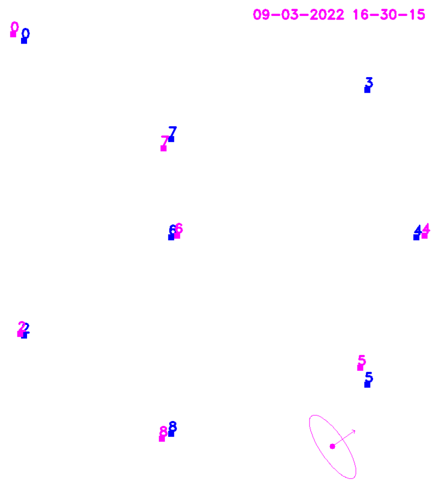

Fig. 1: Sample run of EKF SLAM

1) Parallelize covariance prediction: The parallelization of the covariance leads to four separate functions, *GetTopLeft*, *GetTopRight*, *GetBottomLeft* and *GetBottomRight*. Each one executed in one thread. The number of landmarks for this experiment was set to 10, which mainly influences the size of the matrix computation.

The average time for the multithreaded case averaged to 0.15ms while the sequential run was an order of magnitude lower at 0.0125ms. Varying the number of landmarks was limited by the Eigen C++ library to 100, which made the change in performance barely noticeable for having more or less landmarks.

2) Parallelize measurement prediction jacobian: The measurement prediction jacobian was parallelized by separating different landmark observations into different threads. As the computation is performed in constant time, the runtime reduction is not noticeable without a modification to the code.

A 10ms delay was added to each landmark computation, simulating the process of matching and association in a real world system.

At maximum, six observations were made at the same time by the robot, so having a feature batch size of one is reasonable for an eight core CPU.

Perfroming the operations sequentially lead to the addition of the runtimes for each landmark. If the robot observes 6 landmarks, the delay is 60.3ms, if it observes 2 landmarks, the delay is 20.3ms.

However, parallelization improved this by dividing all the work into multiple threads, thus taking always 10.3ms no matter how many landmarks are seen.

3) Results and discussion: Table I shows the results mentioned earlier, demonstrating the two sides of multithreaded programming. It can be observed how thread creation is expensive. Spawning multiple threads for small problems reduces efficiency by increasing throughput. Meanwhile, computations that require time such as feature matching and detection in a robot SLAM system can be spread across multiple threads. In such a case, an increase in throughput also means an increase in performance.

	Σ_t^- **Computation (ms)**	H_t **Computation (ms)**
Single thread	0.0125	n of observations * 10.3
Multi thread	0.12	10.3

TABLE I: Experiment Results

V. Conclusion

All in all, this report has provided a practical view of how to implement threads and shared memory in C++. It has studied the available options for multithreaded programming with practical examples, and finally demonstrated how it can be used in a robot localization and mapping system. It has also explored how multithreading, apart from requiring good knowledge on implementation to avoid deadlocks and races, can be detrimental if the application is not suitable for it, as well as beneficial for other scenarios. Further research in the topic would include to explore the other concurrent programming areas on atomic operations and abstraction levels.

References

[1] "Multithreading in 20 minutes." [Online]. Available: https://www.youtube.com/watch?v=3aqxaZsvn80

[2] C. Stachniss, "Ekf-slam." [Online]. Available: https://www.youtube.com/watch?v=X30sEgIws0gt=11s

[3] J. Solà, "Simultaneous localization and mapping with the extended kalman filter." [Online]. Available: