TBB Data Flow graphs in Robotics

Nicolo Valigi `nicolovaligi.com`

Italian C++ Day 2019

Self-driving cars

Figure 1: Exercise 1: find me

More down-to-the-carpet projects

Figure 2: Autonomous Roomba

Motivation for this talk

1. Robots are cool! Learn more about them!
2. Most Robotics programming is done in C++
3. A lot of talk lately about `co_await` and coroutines, but little in the way of general architecture for parallelism.

Presentation plan

- The typical architecture of Robotics systems today
- Data flow abstractions for parallelism
- Using data flow graphs in Robotics

Architecture of Robotics systems

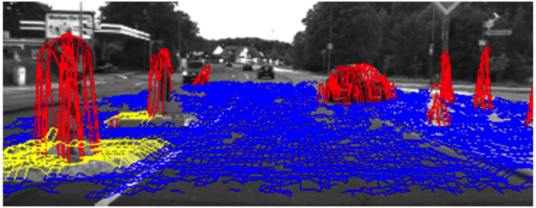
How a robotic system looks like

A typical model for the system architecture of a robot has 3 main components:

- **Sense**
- **Plan**
- **Act**

Sense

Sense is about building a representation of the environment (both fixed obstacles like walls and moving actors like pedestrians).


a)


Plan

Plan is about finding a path to the destination through obstacles and the shape of the environment.

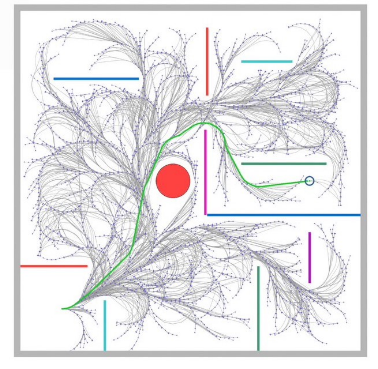


Figure 4: Random graphs used for exploration and planning

Act

Act is about tracking the planned behavior.

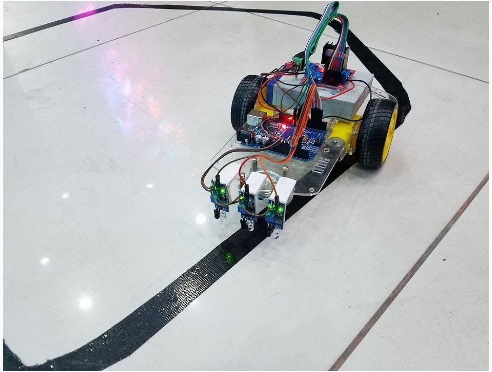


Figure 5: A simple line follower robot

Why Robotics software is fun

- *Sense*, *Plan*, and *Act* need to run concurrently for the robot to work.
- Strict latency and performance requirements

Robotics systems architecture

ROS (Robot Operating System)

About ROS

ROS (Ros Operating System) is the most commonly used middleware for Robotics in use today. It provides:

- serialization and inter-process communication
- high-level tooling for visualization and debugging
- a wealth of code shared by industry and academia

Topics, messages, and subscriptions

The underlying model in ROS is publish/subscribe between independent components.

- Sensor drivers (cameras, etc..) *publish* data over a *topic* (say, `/camera1/image`.
- Computation nodes *subscribe* to messages coming over a topic, do some computation, and produces output.
- Control drivers (steering wheel, brake, etc..) *subscribe* to control outputs and send commands to the mechanical hardware.

The callback model

Subscribe to a specific topic that carries data that we're interested in, do some processing, then publish the result.

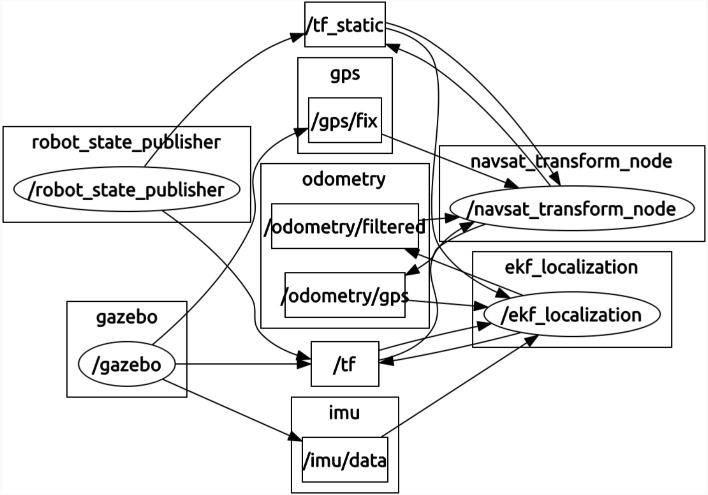
```
// Callback that does the actual processing
void image_callback(const sensor_msgs::ImageConstPtr& msg) {
    std::cout << "Received image" << std::endl;
}

...

// Subscribe to incoming messages on a topic
ros::NodeHandle nh;
auto subscriber = nh.subscribe("/camera1", 1, image_callback);
```

A more complex system

The resulting system is very modular (multiple sensors, layered planning stack, etc..)



Problems with ROS callbacks

However, an architecture based on callbacks doesn't really scale to large architectures:

- it's not clear how data flows through the system
- it's easy to end up using old or stale data in different components
- latency and jitter suffer because of IPC

The Data Flow model

Data Flow graphs

Instead of using callbacks, we can explicitly represent the flow of data throughout the system as a graph.

For example, let's take the simple for loop:

```cpp
// Print the squares of the first 10 squares.
std::vector<int> output;
for (int i = 0; i < 10; i++) {
    const auto thisSquare = i * i;
    std::cout << thisSquare << "\n";
}
```

This is all good and well, but what if we want to run the inner loop in parallel.

A simple Data Flow graph

We can break down the `for` loop as a series of operations where data flows through a graph:

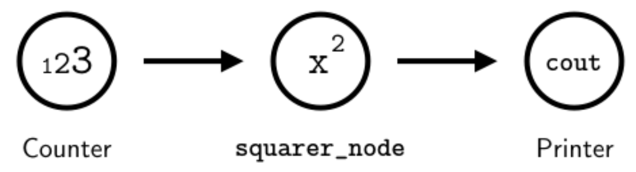


Figure 7: Simple data flow graph

The intel TBB library

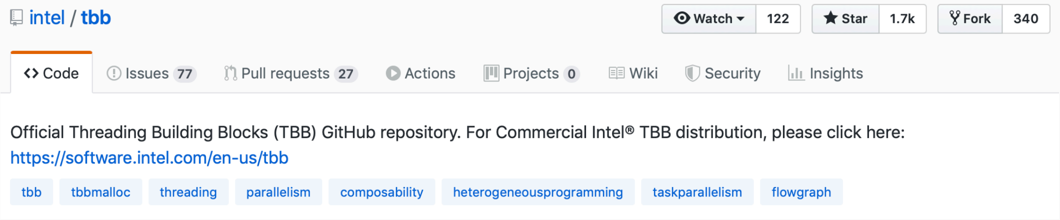


Figure 8: The Intel TBB library

The `function_node`

A `function_node` expresses computation done in the graph (a function, closure, or function object).

```
tbb:function_node<int, int> squarer(g,                         /* the graph */
                                    tbb::unlimited,            /* concurrency */
                                    [](const int &v) {
                                        return v*v;            /* computation */
                                    );
```

The source_node

A source_node pushes data into the graph (generating sequences or reading data from files).

```
struct Counter {
    Counter(int limit) : limit_(limit), i_(0) {}

    bool operator(int& out) {
        if (i < limit_) {
            out = i++; return true;
        } else {
            return false;
        }
    }

    int limit_, i_;
}
```

Plugging the `source_node` into the graph

This takes an instance of the functor and plugs it into the graph where it can start feeding data to other nodes.

```
tbb::flow::source_node<int> counterNode(g, Counter(10), false);
```

The output node

Another `function_node` simply prints out the results of the computation.

```
tbb:function_node<int, int> printerNode(g,
                                        tbb::unlimited,
                                        [](const int &v) {
                                            std::cout << v << "\n";
                                        );
```

Putting it all together

A `tbb:graph` instance manages the connection between the nodes, and can start computation.

```
tbb::graph g;

// Connect the counter node to the computation node.
tbb::make_edge(counterNode, squarerNode);

// Connect the computation node to the output node.
tbb::make_edge(squaredNode, printerNode);

counterNode.activate()

g.wait_for_all();
```

A more complex example

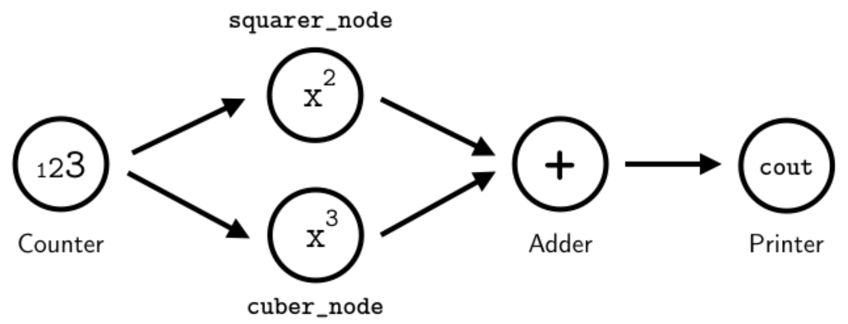


Figure 9: A more complex data flow graph

Data flow in Robotics

QR tags

Figure 10: QR codes are everywhere

QR tags in Robotics

Figure 11: QR codes can also be used for localization

Can we make a graph of this?

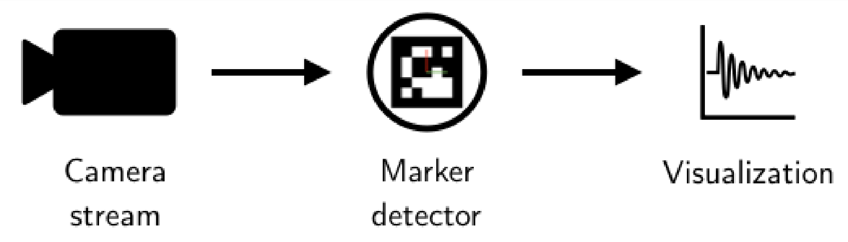


Figure 12: Simple graph for marker detection

This is how it looks like

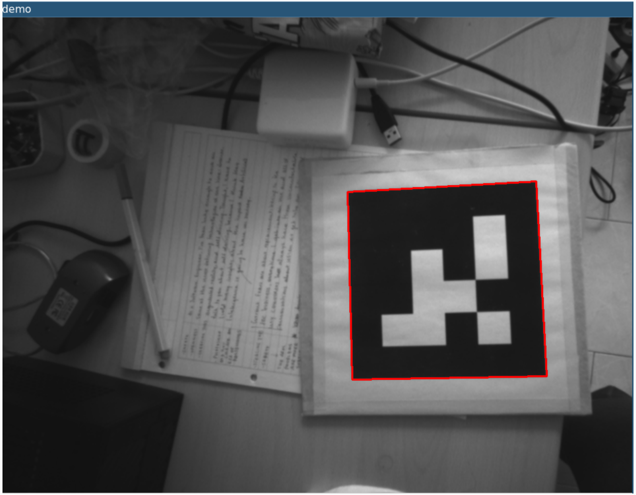


Figure 13: Detected tag

Holding state in a `source_node`

```
struct BagSource {
    BagSource(rosbag::Bag* bag) {
        view_ = std::make_shared<rosbag::View>(*bag);
        it_ = view_->begin();
    }

    bool operator() (sensor_msgs::ImageConstPtr& out) {
        if (it_ == view_->end()) {
            return false;
        } else {
            out = it_->instantiate<sensor_msgs::Image>();
            it_++;
        }
    }
};
```

What about multiple cameras?

Stereo (double) cameras are quite common in Robotics because they add depth perception (just like human eyes).

Figure 14: A cheap stereo camera

Graph layout with multiple cameras

It's trivial to extend a data flow graph to use multiple cameras.

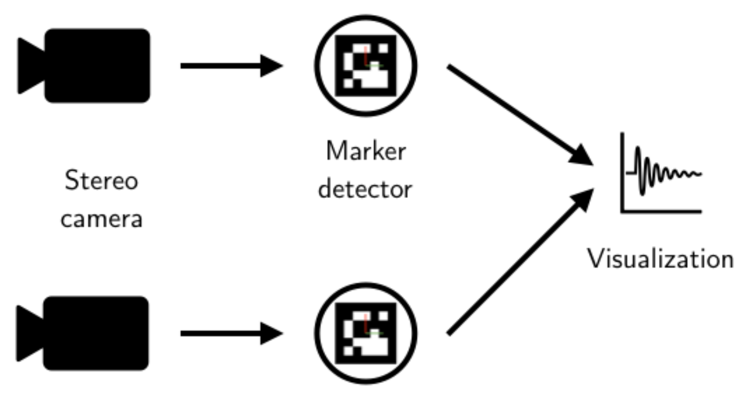


Figure 15: Graph with two cameras

Hooking up external async events

```
struct AsyncRos {
  AsyncRos(ros::NodeHandle &nh) : nh_(nh) {
    sub_ = nh_.subscribe("/device_0/sensor_0/Infrared_1/image/data", 1,
                         &AsyncRos::image_callback, this);
  }
};
```

External async events 2

```
void subscribe(async_ros_node::gateway_type &gateway) {
    gateway.reserve_wait();

    std::thread(&AsyncRos::ros_loop, this, std::ref(gateway)).swap(ros_thread_);
}

void ros_loop(async_ros_node::gateway_type &gateway) {
    ros::spin();
    gateway_->release_wait();
}

void image_callback(const sensor_msgs::ImageConstPtr &img_msg) {
    gateway_->try_put(img_msg);
}
```

Video demo

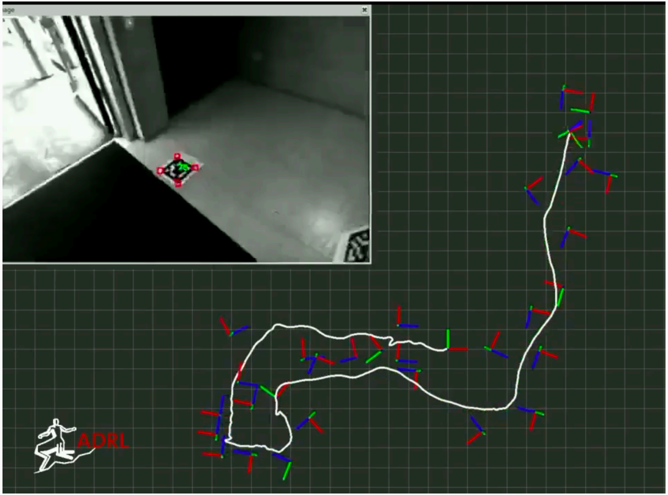


Figure 16: Localization using tags

Tooling

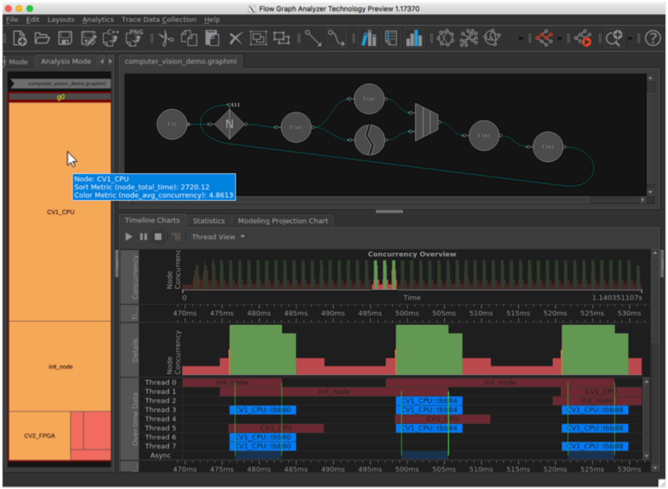


Figure 17: Intel FGA (Flow Graph Analyzer

Advanced topics

- Underlying execution model: *work stealing* scheduler for locality.
- Queueing behavior can be unintuitive at times. In general, you can choose to either buffer on the receiver side, or return the data back to the sender.
- Data passed around the graph is *copied* (inconvenient for large data types)

Alternatives

Many other implementations of Data Flow graphs in C++:

- DSPatch
- TensorFlow
- Pytorch

Why TBB?

- Widely used library
- Integration to SIMD and parallel primitives from the rest of the TBB library.

Thanks / questions