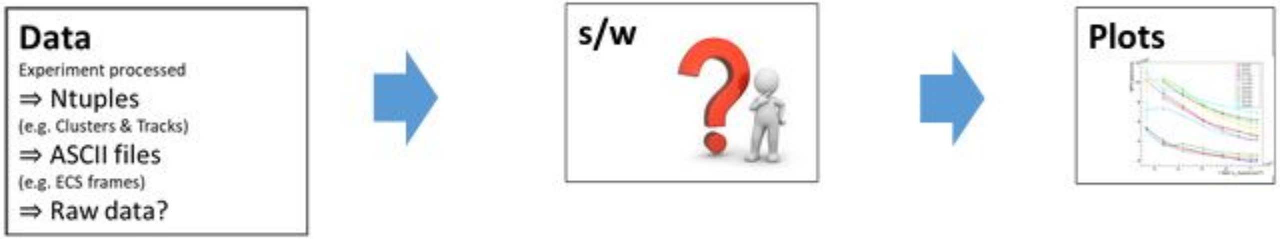
Introducing multithreading mechanisms in ETA framework for event loop data processing

AGH Summer School 2021

Jakub Skrzyński
Project supervisor: Bartek Rachwał

- Framework designed to improve process of building test beam analysis applications
- It processes huge amount of data and with number of parameters to produce results



- Different data type for Tb analyses: e.g.: **.root files** prepared under exact configuration
- **Shared Tb data** to be analysed (e.g. EOS area)
- All Tb analysers **use same preprocessed data**

- Common analysis framework – algorithms and methods to be used by analysers in one place
- Common plotting style, fitting procedures etc. which can be easily adjusted if necessary

- Easy compilable and reproducible Tb results
- Series of control plots
- Systematics

Data Analysis: The problem

- Much data, many parameters
- Whole process consisting of: filtering data, filling histograms, fitting and analysing of final cumulative plots
- Event loop stage: filtering data, filling histograms

Concurrency…? Why to do many things at the same time?

- Requires less time
- Better utilizes the resources
- Filtering events does not require sequential processing - it may be analyzed separately

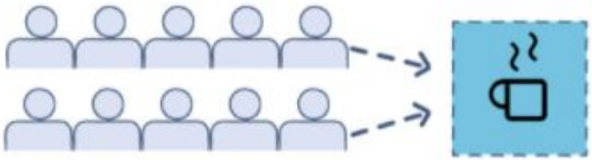
How to approach multi-tasking?

- Concurrent
- Parallel

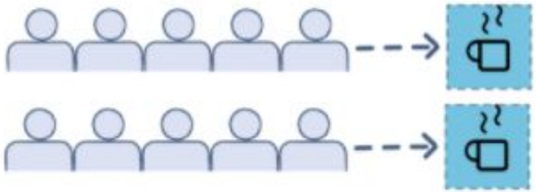
Concurrent way requires us to sometimes lock object for modification

Parallel execution is done through separate instances of program that do not interfere with each other

Concurrent: *Two Queues* & a *Single* Espresso machine.



Parallel: *Two Queues* & *Two* Espresso machines.



- RDataFrame (advised way)

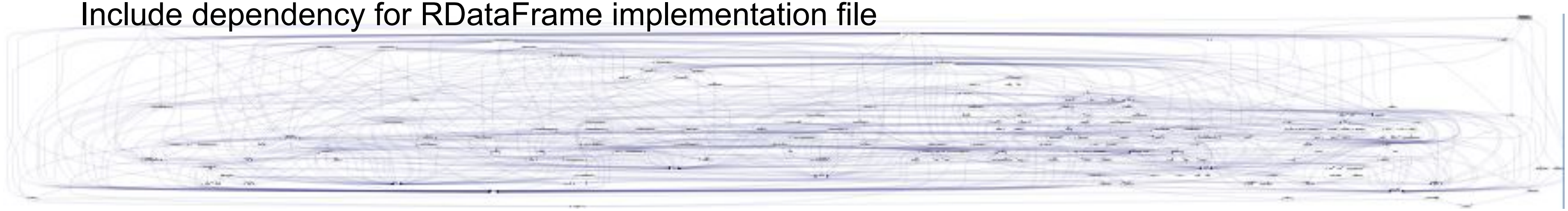
```
ROOT::EnableImplicitMT(); // Enable ROOT's implicit multi-threading
ROOT::RDataFrame d("myTree", "file_*.root"); // Interface to TTree and TChain
auto histoA = d.Histo1D("Branch_A");  // Book the filling of a histogram
auto histoB = d.Histo1D("Branch_B");  // Book the filling of another histogram
// Data processing is triggered by the next line, which accesses a booked result for the first time
// All booked results are evaluated during the same parallel event loop.
histoA->Draw(); // <-- event loop runs here!
histoB->Draw(); // HistoB has already been filled, no event loop is run here
```

- TSelector with the driver
  - PROOF
  - TTreeProcessorMP (currently causes crash)
- TProcessExecutor (issue of shared objects)
- Own implementation (std::thread)

Why not the RDataFrame?

- It is final product
- The class is difficult to extend and wrap
- Complicated
- Complex

Include dependency for RDataFrame implementation file



TSelector

- Allows to easily traverse the TTree
- Enables user to easily access the nTuple data
- Base for multithreading provided by ROOT
- Easy & clear to use interface

Why framework?

- Main goal is to achieve easy way to develop analysis
- Frameworks do set regulations on way of coding – using framework has huge impact on how the code looks like

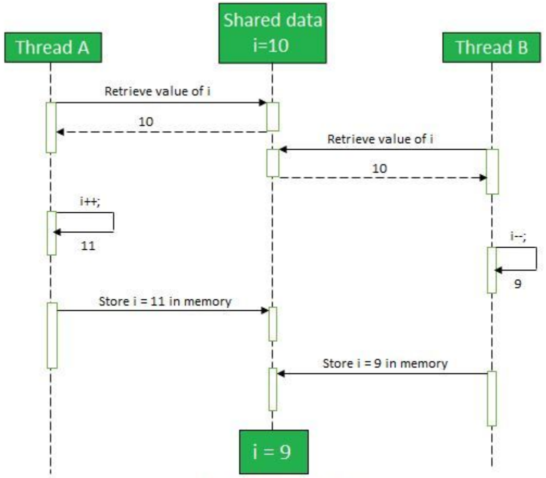
- It derives from TSelector provided by user
- Enables easy preparation of processing procedure

```
template <typename T>
class EtaSelector: public T{
private:
    std::function<void(EtaSelector<T>*)> processor;            //Function that does actual processing
    std::vector<std::function<bool(EtaSelector<T>*)>> filters; //Functions used to filter the events
public:
   //Process -> Implementation of function provided in TSelector. It is called whenever entry is processed
    Bool_t Process (Long64_t entry) override;
   //Init -> Implementation of function provided in TSelector. It is responsible for setting up the tree
    void Init(TTree *tree) override;
   //Implementation of function provided in TSelector
    Bool_t Notify() override;
   //Default constructor of EtaSelector
    EtaSelector()=default;
   //Function used to set the processing function
    void SetFunction(std::function<void(EtaSelector<T>*)>  func){processor=func;}
   //Function used to register filters
    void AddFilter(std::function<bool(EtaSelector<T>*)> filter){filters.push_back(filter);}
};
```

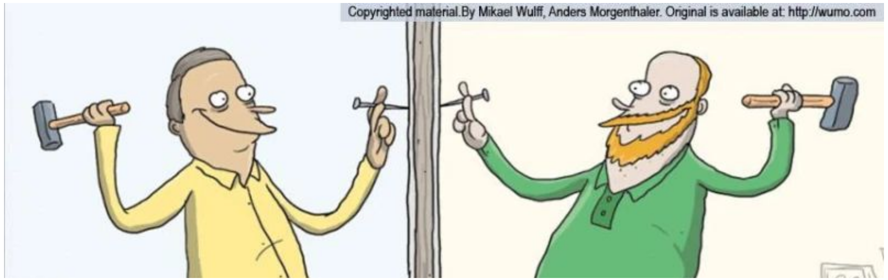
- One can use ROOT provided functionalities such as PROOF, but it seems it is not best solution in this case
- One can create multiple threads with use of implemented our ETA Framework STL based mechanism

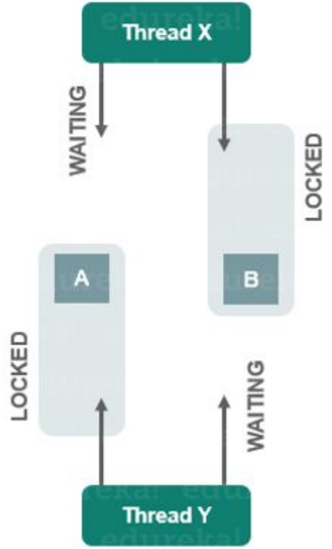
Multithreading

- If multiple threads access the same variable objects, we have to care about concurrency. This is suboptimal solution as locking histograms may cause delays and increase processing time.
- I have decided to clone varying objects, so each thread has separate one



Result of thread A is lost





Multithreading - implemented architecture

```
template<typename T>
class MasterThread{
private:
    //Number of threads to be created
    int threadNumber;
    //Pointer to TTree - input data
    TTree * tree;
    //Map of histograms that will be used while processing
    std::map<std::string,TH1 *> TH1List;
    //Function used to clone one dimmensional histograms for a thread
    std::map<std::string,TH1 *> CloneTH1ForThread();
    //Function that merges histograms received from threads into the final result
    void mergeTH1(std::map<std::string,TH1*> cloned);
    //Variable that holds function that does actual processing
    std::function<void(EtaSelector<T>*,std::map<std::string,TH1 *>)> processor;
    //List of functions used to filter the events
    std::vector<std::function<bool(EtaSelector<T>*)>> filters;
public:
    MasterThread(int threads,TTree * tree):threadNumber(threads),tree(tree){}
    //Function that registers histograms for later use inside thread
    void registerHistogram(TH1 * hist, std::string key){TH1List[key]=hist;}
    //Function that creates threads
    bool Run();
    //Function used to set the processing function
    void SetFunction(std::function<void(EtaSelector<T>*,std::map<std::string,TH1 *>)>  func){processor=func;}
    //Function used to register filters
    void AddFilter(std::function<bool(EtaSelector<T>*)> filter){filters.push_back(filter);}
};
```

Multithreading - implemented architecture

```
template<typename T>
class WorkerThread:public EtaSelector<T>{
private:
    std::thread * th;           //Pointer to STL thread object
    TTree * tree;               //Pointer to Ttree
    int threadId;               //Id of current thread
    int threadNumber;           //Number of all threads
    static int nextThreadId;    //Variable that stores id for next thread

public:
    WorkerThread(std::map<std::string,TH1 *> histogram, TTree * tree, int threadNumber):tree(tree),threadNumber(threadNumber){
        EtaSelector<T>::Init(tree);
        EtaSelector<T>::PassHistograms(histogram);
        threadId = (WorkerThread<T>::nextThreadId++);
    }
    std::map<std::string,TH1 *> RetriveResults(){return EtaSelector<T>::RetriveHistograms();}
    void Run();                                 //Driver function that starts the thread
    void InThreadFunction();                    //Function that is run inside the thread
    std::thread * GetThreadPtr(){return th;}    //Returns pointer to STL thread object
};
```

How to use the code? - example implementation

```
// Preparing TTree
TFile *F = new TFile("ntuple_file.root");
TDirectoryFile *tdf;
F->GetObject("TbTupleWriter", tdf);

TTree *t;
tdf->GetObject("Clusters", t);

//Setting up the analysis
MasterThread<BaseSelector>* mt = new MasterThread<BaseSelector>(1,t);
TH1F * hist;
mt->registerHistogram(hist,"h1");
mt->SetFunction(FillHist);
mt->AddFilter(filter1);
bool done = mt->Run();
```

```
void FillHist(EtaSelector<BaseSelector> *s,std::map<std::string,TH1 *> m){
    m["h1"]->Fill(*(s->clCharge));
}
bool filter1(EtaSelector<BaseSelector> *s){
    return true;
}
```

Plans for future

- Develop complete toolset for threads management
  - Worker process class
  - Master process class
- Test all components and ensure that all cases are covered
- (*) Further development to allow PROOF processing (Motivated by whole analysis preservation idea)

Thank you for your attention!