Framework for distributed alignment

Introduction

In Run 2 of the LHC it will be necessary to quickly (~minutes) (at least) verify the alignment of the detector for the HLT trigger at the beginning of each fill. For this purpose the HLT farm infrastructure will be used to analyse a number of selected events on each node and combine the results from each node in a task that will determine an updated set of alignment constants. This process is repeated until convergence is reached.

The distributed nature of the alignment evidently needs a framework for coordinating the different activities (analysis of the data and the determination of the alignment constants and iterating until convergence is reached). This pamphlet will explain the process and the collaborating components and the framework, especially the 'protocols' to follow.

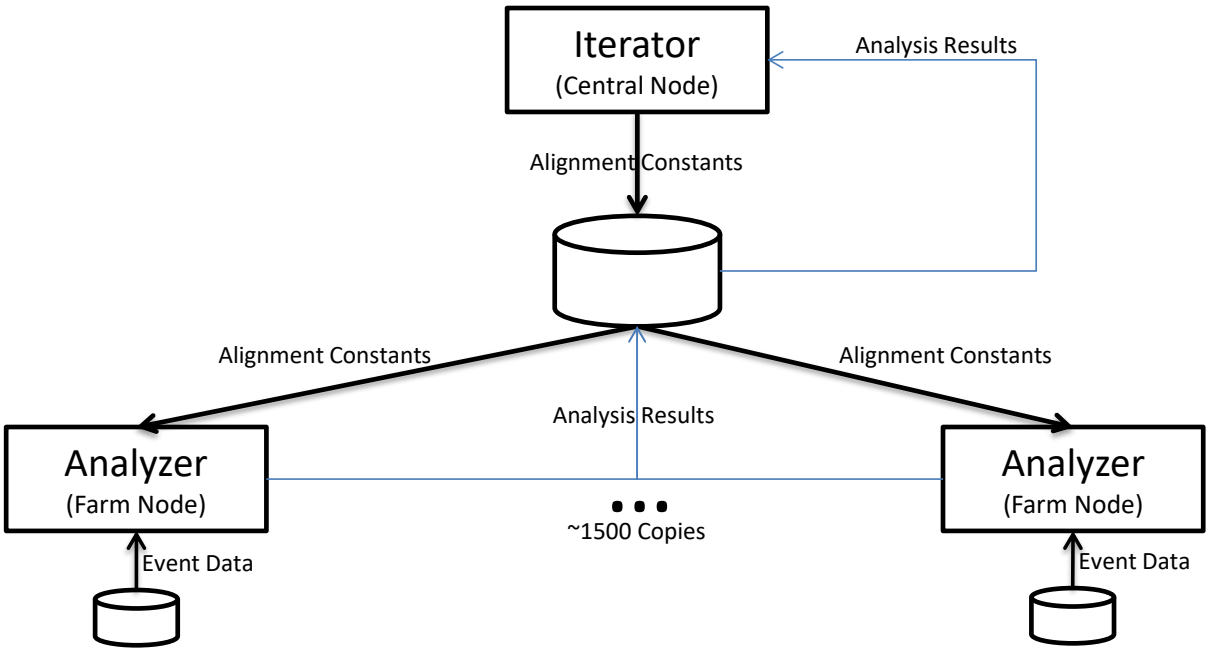

Figure 1 Architecture of the Alignment System

Each task in the system must follow the following state diagram. The transitions depicted in Figure 2 State diagram each task in the system has to follow. The transitions are (in general) invoked by the run controller. There are, however, some transitions that are initiated by the task itself and forwarded to the control system.

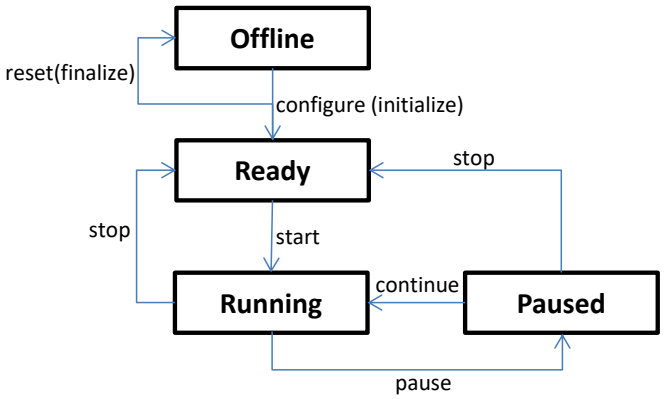

Figure 2 State diagram each task in the system has to follow. The transitions are (in general) invoked by the run controller.

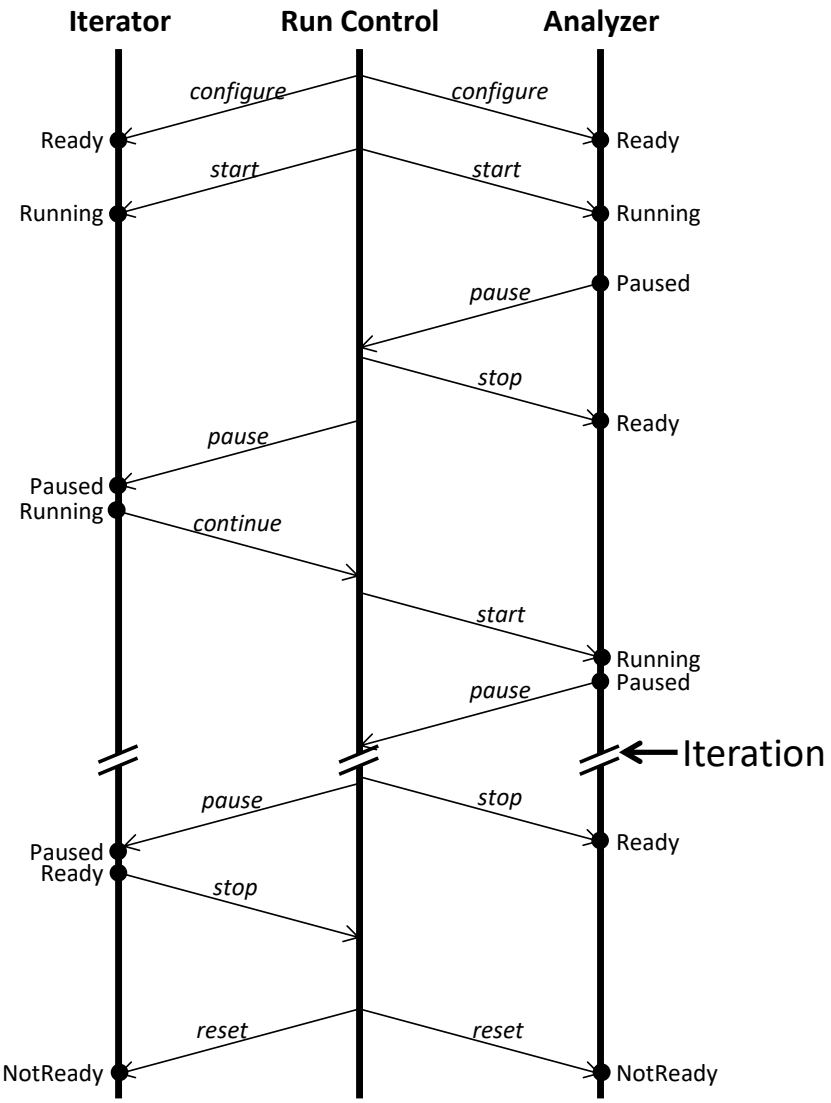

Figure 3 Time sequencing of the alignment process, showing the main components (Iterator, Run Controller, Analyzer). *Italics* denote transitions, straight denote States.

Figure 3 shows the time sequencing of the Alignment process.

In the following sections we outline the obligations of the Iterator and the Analyzer task(s) in the various transitions and states.

Task Obligations

Iterator

Transitions

Configure (initialize() transition in Gaudi)

During this transition the Iterator has to write the initial version of the parameter file to a global storage (typical somewhere in the group area). The filename should be a property and has to be the same for the Analyzers. In addition, obviously, all other initialisations have to be performed.

Start

In this transition the iteration function is called. If, for example, the iteration process involves a 'simple' Chi2 minimisation, the Migrad method of TMinuit would be called. It is important to note that (in general) the iteration function does relinquish control before the iteration process has succeeded (converged) or failed. As a consequence, the working function (e.g. the Chi2 function) has to be executed in a separate thread, so that state transitions can still be executed or initiated. However, the working function can only provide the iteration function with a new value, once the Analyzing processes have provided with their results, signalled by the run controller by initiating the *pause* transition. In a later section these aspects will be discussed in-depth.

Pause

In this transition the iterator has to collect the results from the worker functions and pass it to the iterating subsystem (e.g. Migrad in Minuit or the MVA master) to determine the next parameter set.

The *pause* transition is initiated by the run control system signalling that all worker processes have finished the evaluation with the current parameter set.

Continue

The *continue* transition is initiated by the Iterator and signals to the run controller, and subsequently to the worker processes on the farm, that a new set of parameter values is available.

During the iteration process the Iterator will oscillate between the *running* and *paused* state.

Analyzer(s)

There are two flavours of analyzers

- Analyzers that use event data (from files via an event selector) to determine the iteration value
- Analyzers that use any other way of determining the iteration value

Transitions

Configure (initialize() transition in Gaudi)

During this transition the task initializes all internal values and prepares for the subsequent work.

Start

During this transition the Analyzer task reads the parameter set that has been provided by the iterator. After this transition the task is in the *running* state and the framework will call the *run* method to launch the event processing loop (typically reading event data from file and call the *execute* method. Once the end of the data file is reached, the analyser task (actually the event selector) will initiate the *pause* transition.

Pause

This transition signals the run controller that the Analyzer task has finished processing the events and has the results ready for publication. This transition is initiated by the event selector upon reaching end-of-file on the event data.

Stop

The stop transition is initiated by the run controller to trigger the publication of the analysis results for the iterator to collect. The combined analysis results will be fed to the iteration subsystem to provide the next parameter set to be evaluated.

During the iteration process, the analyzers will cycle around the *running→paused→ready→running* states.

Support Framework for writing Analyzers and Iterators

There are two support components that should facilitate writing Analyzers and Iterators available.

- AlignDrv for implementing an Iterator (Driver process) of the alignment procedure
- AlignWork for implementing an Analyzer (Work process)

It should be noted that analyzers that use event data, read from files using a standard event selector, do not need to implement specific code.

We will, thus, in the following focus on the Alignment Driver (Iterator).
The Iterator is based on the Gaudi *AlgTool* and has to implement the *IAlignIterator* interface.

```
class IAlignIterator : virtual public IAlgTool
{
  public:
    DeclareInterfaceID(LHCb::IAlignIterator,1,0);
    virtual StatusCode i_start()=0;
};
```

The *i_start()* method is called from the parent Gaudi Service that handles all the communication with the external world (Application manager/Run Controller) in the start transition.

The parent Gaudi Service implements the following interface that can (has to) be used by the iterator.

```
  class GAUDI_API IAlignDrv: virtual public IInterface
  {
    public:
      DeclareInterfaceID(IAlignDrv,1,0);
      virtual void writeReference()=0;
      virtual void waitRunOnce()=0;
      virtual void doContinue()=0;
      virtual void doStop()=0;
  };
```

The header file for the Fitter (Iterator) is listed in the following section

```
#ifndef ONLINE_GAUCHO_FITTER_H
#define ONLINE_GAUCHO_FITTER_H

#include "GaudiKernel/Service.h"
#include "GaudiKernel/IToolSvc.h"
#include "RTL/rtl.h"
#include "GaudiKernel/AlgTool.h"
#include "IAlignUser.h"
#include "IAlignSys.h"

// Forward declarations
class TMinuit;
class CounterTask;
namespace LHCb
{
  class Fitter : public AlgTool, virtual public LHCb::IAlignIterator
  {
    public:
      Fitter(const std::string &  type, const std::string &  name, const IInterface
*  parent  );
      TMinuit *m_Minuit;
      IAlignDrv *m_parent;
      StatusCode i_start();
      StatusCode i_run();
      StatusCode initialize();
      StatusCode finalize();
      StatusCode stop();
      void write_params(int,std::vector<double> &params);
      void write_params(int npar, double *params);
      void read_params(int&,std::vector<double> &params);
      double getIterationResult();
      lib_rtl_thread_t m_thread;
      CounterTask *m_cntTask;
      std::string m_ParamFileName;
      std::vector<std::string> m_CounterNames;
      std::string m_CntDNS;
      std::string m_PartitionName;
      std::string m_CntTask;
      std::vector<double> m_params;
      StatusCode queryInterface(const InterfaceID& riid, void** ppvIF);
  };
}
#endif // ONLINE_GAUCHO_FITTER_H
```

In the following paragraph the implementation of the Fitter class is listed.

```
#include <stdio.h>
#include <stdlib.h>
#include "Fitter.h"
#include "Gaucho/CounterTask.h"
#include "Gaucho/MonCounter.h"
#include "TMinuit.h"
DECLARE_NAMESPACE_TOOL_FACTORY(LHCb,Fitter)

using namespace LHCb;
static Fitter *FitterInstance;

extern "C"
{
  int FitterThreadFunction(void *t)
  {
    Fitter *f = (Fitter*)t;
    FitterInstance = f;
    f->i_run();
    return 1;
  }
}
```

The FitterThreadFunction implements the thread code that executes the master iterator.
This functionality has to be implemented in a separate thread, as the Iterator task still has to follow the transitions initiated by the run controller.

```
  void Chi2(int &npar, double */*grad*/, double &fval, double *params, int
/*flag*/)
  {
    FitterInstance->m_parent->waitRunOnce();
    fval = FitterInstance->getIterationResult();
    FitterInstance->m_parent->writeReference();
    FitterInstance->write_params(npar,params);
    FitterInstance->m_parent->doContinue();
    return;
  };
};

Fitter::Fitter(const std::string &  type, const std::string &  name, const
IInterface *  parent  ):AlgTool(type,name,parent)
{
  declareProperty("PartitionName",   m_PartitionName= "LHCbA");
  declareProperty("ParamFileName",   m_ParamFileName= "/home/beat/aligparams.dat");
  declareProperty("CounterNames",m_CounterNames={"aaa/Chi2"});
  declareProperty("CounterDNS",m_CntDNS="mona08");
  declareProperty("CounterTask",m_CntTask="LHCbA_AligWrk_00");
  m_Minuit = 0;
  IInterface *p=(IInterface*)parent;
  StatusCode sc = p->queryInterface(IAlignDrv::interfaceID(),(void**)(&m_parent));
  m_cntTask=0;
}

StatusCode Fitter::initialize()
{
  int npar;
  m_Minuit = new TMinuit(3);
  m_Minuit->SetFCN(&Chi2);
  read_params(npar,m_params);
  for (size_t i=0;i<m_params.size();i++)
  {
    char nam[2];
    char ii;
    ii = char(i);
    nam[0] = 'a'+ii;
    nam[1] = 0;
    m_Minuit->DefineParameter(i,nam,m_params[i],10.0,0.0,0.0);
  };
  return StatusCode::SUCCESS;
}

StatusCode Fitter::i_run()
{
  int res=m_Minuit->Migrad();
  double par[100],dpar[100];
  printf("MIGRAD has finished with return code: %d\nParamters:\n",res);
  for (unsigned int i=0;i<m_params.size();i++)
  {
    m_Minuit->GetParameter(i,par[i],dpar[i]);
    printf("Param #%d %15g +- %15g\n",i,par[i],dpar[i]);
  }
  fflush(stdout);
  m_parent->doStop();
  return StatusCode::SUCCESS;
}

StatusCode Fitter::i_start()
{
  FitterInstance->m_parent->writeReference();
  FitterInstance->write_params(npar,params);
  ::lib_rtl_start_thread(FitterThreadFunction,this,&m_thread);
  return StatusCode::SUCCESS;
}

StatusCode Fitter::stop()
{
  ::lib_rtl_delete_thread(m_thread);
  return StatusCode::SUCCESS;
}

StatusCode Fitter::finalize()
{
  if (m_Minuit)
  {
```

The sequence of these calls has to be maintained

The Chi2 function is the provider of iteration data to the Iteration master routine (e.g. Migrad in the case of Minuit). It will be called by the Iteration master whenever a new set of parameters is available.

These two lines are mandatory

This call is mandatory

These calls are mandatory

This call is mandatory

```
    delete m_Minuit;
    m_Minuit = 0;
  }
  return StatusCode::SUCCESS;
}

double Fitter::getIterationResult()
{
  std::vector<CntrDescr*> cdesc;
  if(m_cntTask == 0)
  {
    m_cntTask = new CounterTask(m_CntTask,m_CntDNS);
  }
  m_cntTask->Counters(m_CounterNames,cdesc);
  if (cdesc.size() > 0)
  {
    return cdesc[0]->d_data;
  }
  else
  {
    printf("No Counters Found...\n");
  }
  return nan("");
}

void Fitter::write_params(int npar, std::vector<double> &params)
{
  FILE *f;
  f = fopen(m_ParamFileName.c_str(),"w");
  for (int i=0;i<npar;i++)
  {
    fprintf(f,"%15lf ",params[i]);
  }
  fprintf(f,"\n");
  fclose(f);
}

void Fitter::write_params(int npar, double *params)
{
  FILE *f;
  f = fopen(m_ParamFileName.c_str(),"w");
  for (int i=0;i<npar;i++)
  {
    fprintf(f,"%15lf ",params[i]);
  }
  fprintf(f,"\n");
  fclose(f);
}

void Fitter::read_params(int &npar, std::vector<double> &params)
{
  FILE *f;
  params.clear();
  f = fopen(m_ParamFileName.c_str(),"r");
  while (!feof(f))
  {
    double p;
    fscanf(f,"%lf",&p);
    if (feof(f)) break;
    params.insert(params.end(),p);
  }
  fclose(f);
  npar = params.size();
}

StatusCode Fitter::queryInterface(const InterfaceID& riid, void** ppvIF)
{
  if (LHCb::IAlignIterator::interfaceID().versionMatch(riid))  {
    *ppvIF = (IAlignIterator*) this;
    addRef();
    return StatusCode::SUCCESS;
  }
  return AlgTool::queryInterface(riid, ppvIF);
}
```

In this routine retrieve the results from the Worker tasks in the farm. This routine is called, once all tasks on the farm nodes have finished processing

Write the parameters to a globally accessible file that can be subsequently read by the worker tasks on the farm

Write the parameters to a globally accessible file that can be subsequently read by the worker tasks on the farm