37th LHCb Software Week

Pere Mato (CERN), Eoin Smith (CERN)

Motivation

- Work done in the context of the R&D project to exploit multi-core architectures
- Main goal
  - Get quicker responses when having 4-8 core machines at your disposal
  - Minimal changes (and understandable) to the Python scripts driving the program
- Leverage from existing third-party Python modules
  - *processing* module, *parallel python* (pp) module
- Optimization of memory usage

A typical GaudiPython script

```
#--- Common configuration -------------------------
from Gaudi.Configuration import *

importOptions('$STDOPTS/LHCbApplication.opts')
importOptions('DoDC06selBs2Jpsi2MuMu_Phi2KK.opts')

#--- modules to be reused in all processes----------
from GaudiPython import AppMgr
from ROOT import TH1F,TFile,gROOT

files = [... ]
hbmass = TH1F('hbmass','Mass of B cand',100,5200.,5500.)
appMgr = AppMgr()
appMgr.evtsel().open(files)
evt = appMgr.evtsvc()

while True :
  appMgr.run(1)
  if not evt['Rec/Header'] : break
  cont = evt['Phys/DC06selBs2Jpsi2MuMu_Phi2KK/Particles']
  if cont :
    for b in cont : hbmass.Fill(b.momentum().mass())

hbmass.Draw()
```

Predefined configuration + customizations

Preparations (booking histos, selecting files, etc.)

Looping over events

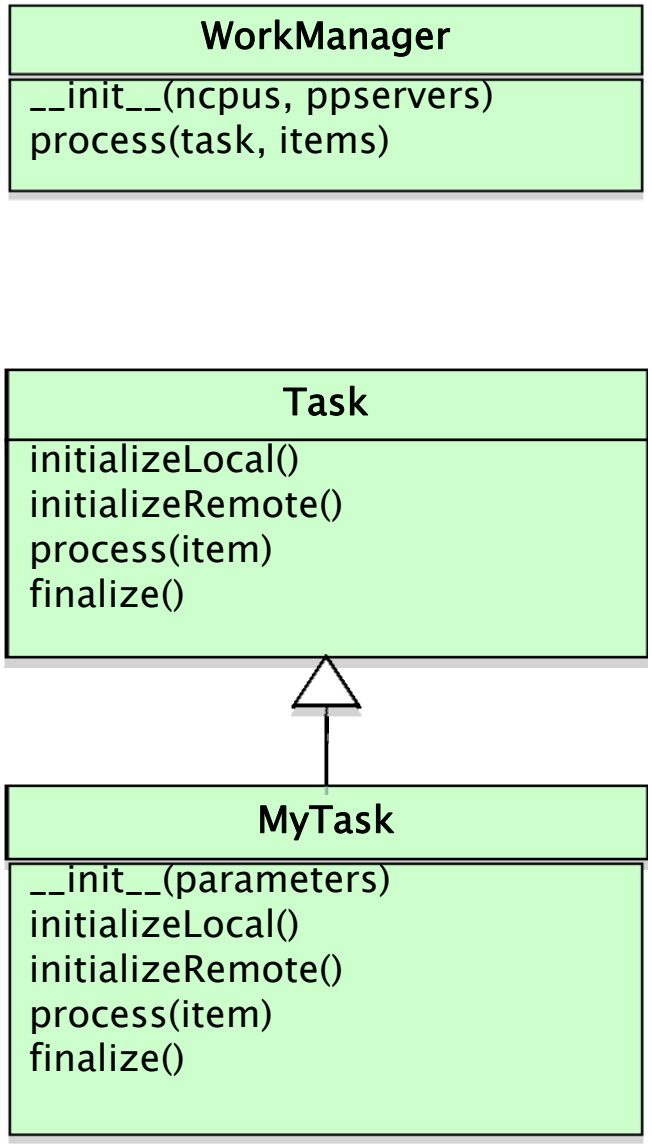
Presenting results

Parallelization Model

- Introduced a very simple Model for parallel processing of tasks
- Common model that can be implemented using either *processing* or *pp* modules (or others)
- The result of processing can be any 'pickle-able' Python or C++ object (with a dictionary)
- Placeholder for additional functionality
  - Setting up the environment (including servers)
  - Monitoring
  - Merging results (summing what can be summed or appending to lists)

The Model

- User parallelizable task derives from *Task*
  - *initializeLocal*() is executed in parent process
  - *initializeRemote*() is executed once in each remote process
  - *process*() is executed for each work *item* in remote process
  - *finalize*() is executed at the end in the parent process

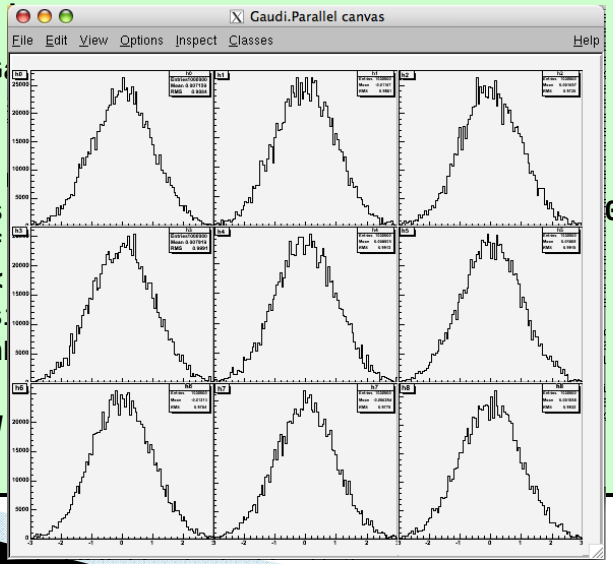


Very simple example

```
from ROOT import TH1F, TRandom, TCanvas, gROOT
from GaudiPython.Parallel import Task, WorkManager

class HistTask(Task):
  def __init__(self, nHist=4) :
    self.nHist = nHist
    self.canvas = None

  def initializeLocal(self):
    self.output = [TH1F('h%d'%i,'h%d'%i,100,-3.,3.) for i in range(self.nHist)]
    self.random = TRandom()

  def in
  def pr
    for h in self.output :
      for i in range(n):
        x = self.random.G
        h.Fill(x)

  def finalize(self):
    self.canvas = TCanvas
    nside = int(sqrt(self
    nside = nside*nside <
    self.canvas.Divide(ns
    for i in range(self.n
      self.canvas.cd(i+1)
      self.output[i].Draw
```

```
>>> from GaudiPython.Parallel import WorkManager
>>> from HistTask import HistTask

>>> task = HistTask(nHist=9)
>>> wmgr = WorkManager()
>>> wmgr.process( task, [100000 for i in range(100)])
```

```
Job execution statistics:
job count | % of all jobs | job time sum | time per job | job server
      100 |        100.00 |      378.296 |       3.783  | lxbuild114.cern.ch
Time elapsed since server creation 60.805200
```

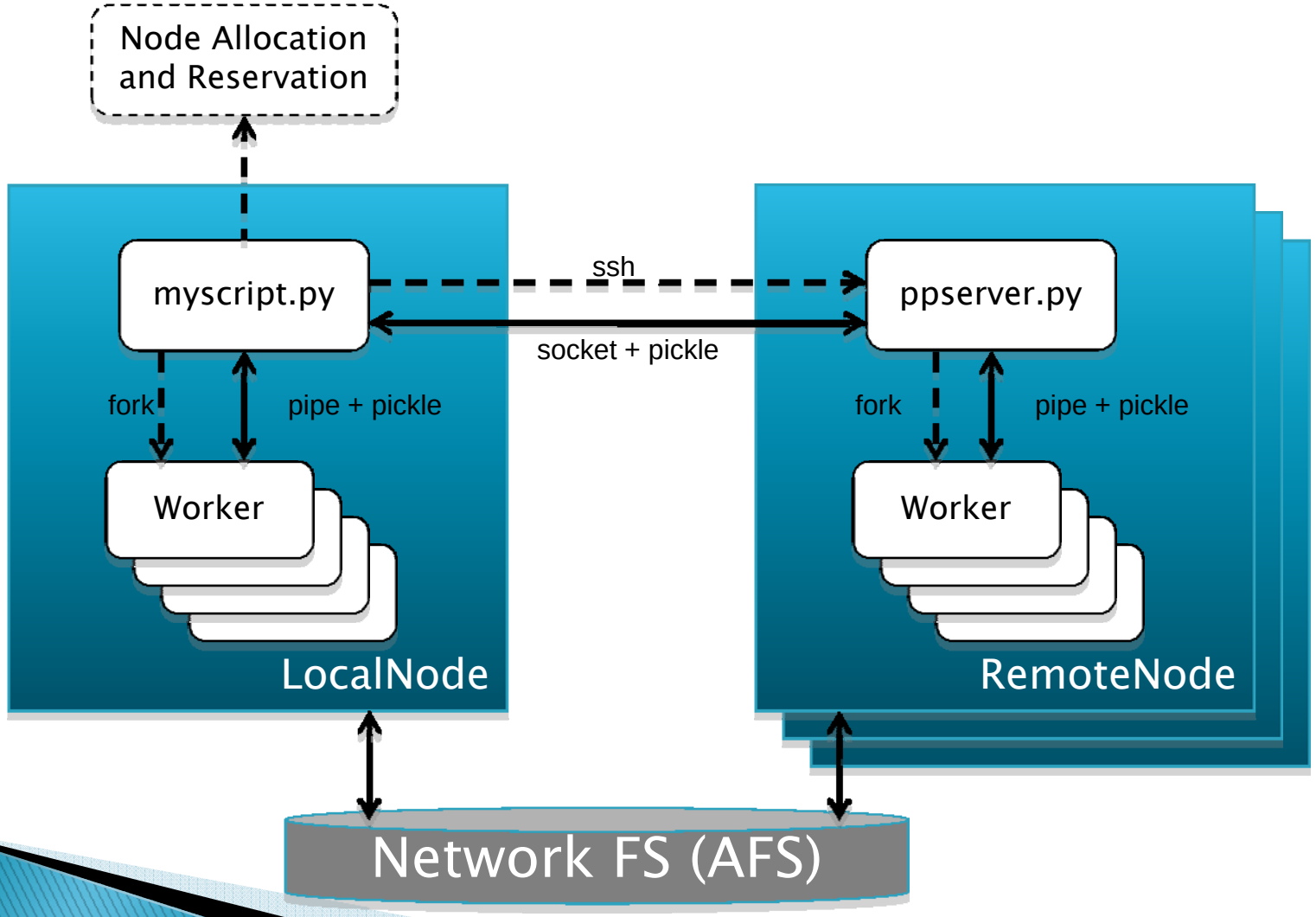


More Realistic Example

```
from ROOT import TFile,TCanvas,TH1F,TH2F
from Gaudi.Configuration import *
from GaudiPython.Parallel import Task, WorkManager
from GaudiPython import AppMgr

class MyTask(Task):
  def initializeLocal(self):
    self.output = {'h_bmass': TH1F('h_bmass','Mass of B candidate',100,5200.,5500.)}
  def process(self, file):
    appMgr = AppMgr()
    appMgr.evtsel().open([file])
    evt = appMgr.evtsvc()
    while 0 < 1:
      appMgr.run(1)
      # check if there are still valid events
      if evt['Rec/Header'] == None : break
```

```
from GaudiPython.Parallel import WorkManager
from MyTask import MyTask
from inputdata import bs2jpsiphimm_files as files

ppservers = ('lxbuild113.cern.ch','lxbuild114.cern.ch')

if __name__ == '__main__' :
  task = MyTask()
  wmgr = WorkManager(ppservers=ppservers)
  wmgr.process( task, files[:50])
```

Parallelization on a Cluster



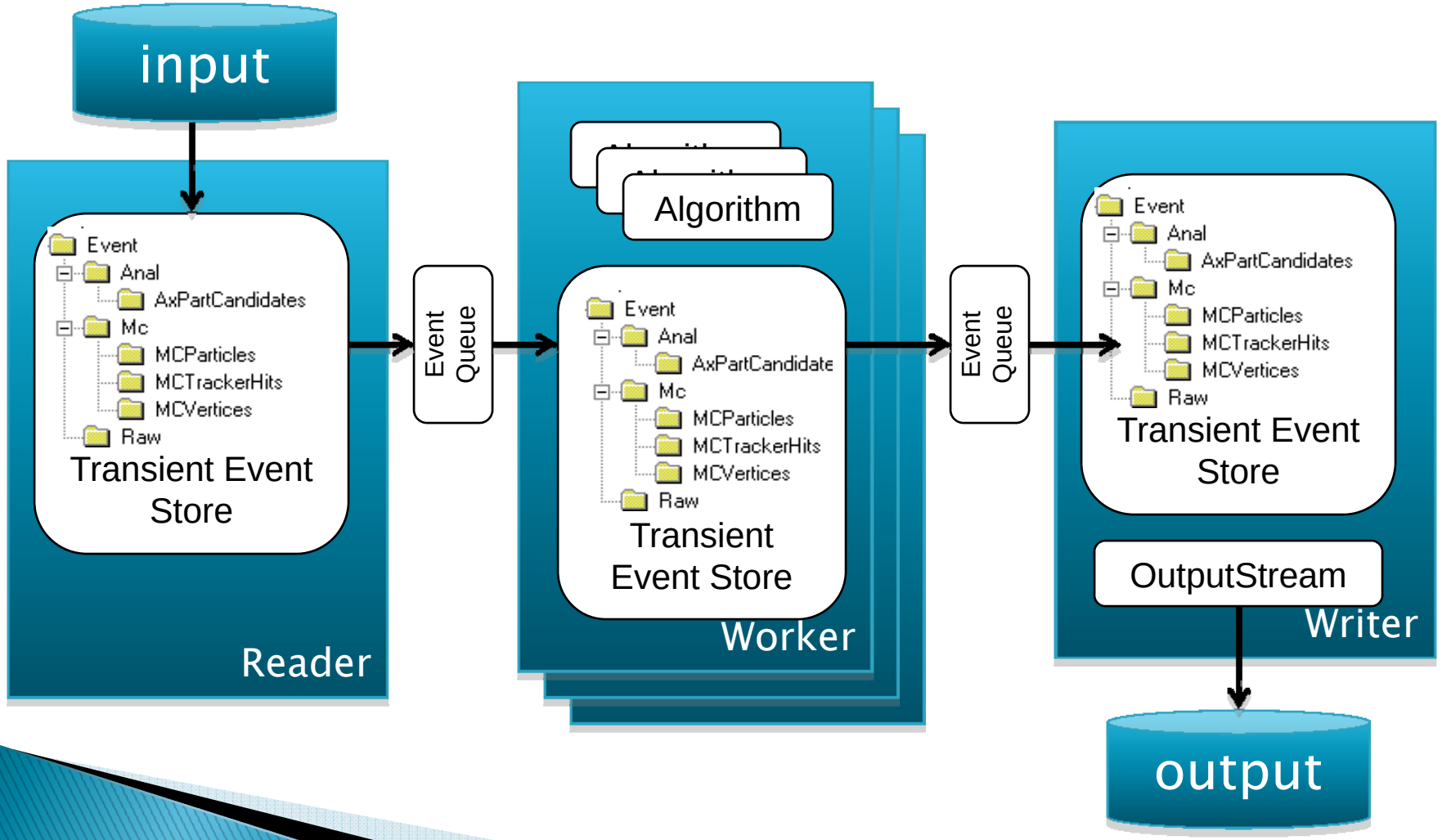
Cluster Parallelization Details

- The [pp]server is created/destroyed by the user script
  - Requires ssh properly setup (no password required)
- Ultra-lightweight setup
- Single user, proper authentication, etc.
- The ‘parent’ user environment is cloned to the server
  - Requires a shared file system
- Automatic cleanup on crashes/aborts

Model fails for “Event” output

- In case of a program producing large output data the current model is not adequate
  - Event processing programs: simulation, reconstruction, etc.
  - Parallelization at event level
- Two options:
  - Each sub-process writing a different file, parent process merges the files
    - Merging files non-trivial when “references” exists
  - Event data is collected by one single process and written into a single output stream
    - Some additional communication overhead

Handling Event Input/Output



User Interface in development

```
gaudirun --parallel=N optionfile.py
```

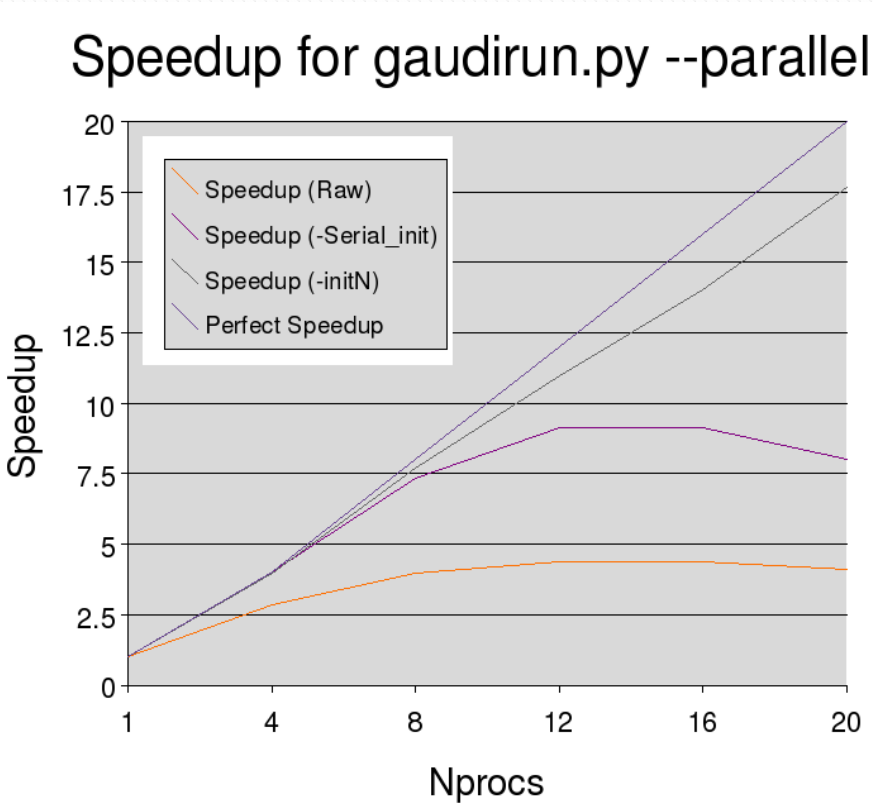
- No modification needed in user options file
- Modified version of *gaudirun.py* script that will be hiding all the details
  - Sub-processes are spawned with slightly modified set of options

Handling Input/Output

- Developed event serialization/deserialization (TESSerializer)
  - The entire transient event store (or parts) can be moved from/to sub-processes
  - Implemented by pickling a TBufferFile object
  - Links and references must be maintained
- Current status
  - Completed for “event data”
  - Working on “statistical data” (histograms)
  - Problematic for Ntuple
  - More work and definition needed for counters/run records

Preliminary Results

- --parallel=N tried with Brunel v35r1 (1000 events) in a 24 core machine
  - Initialization rather large compared to total time
  - Subtracted to get more realistic with large # events



Brunel v35r1

Nprocs	1	4	8	12	16	20
$T_{initNprocsevts}$	46.83	48.44	51.57	55.54	61.77	71.46
$T_{init1evt}$	46.83†	46.75†	50.58†	51.76†	57.08†	64.73†
$T_{1000evts}$	360.84	128.95	93.2	84.72	84.58	89.5

Status

- Current version of Gaudi includes GaudiPython.Parallel
  - Ready and useful for the file-level parallelization
- The event-level parallelization is going to be made available as soon is more complete
  - At least Histograms should be available
  - Validation of ‘serial’ vs ‘parallel’ results
- Further work
  - Measure realistic ‘speedup’ with longer jobs
  - Optimizing Inter-process-communication
  - Forking ‘workers’ after initialization to save memory

Summary

- Very simple adaptation of Python scripts may led to huge gains in latency (multi-core)
  - Already usable today
- Several schemas and scheduling strategies can be tested without changes in the C++ code
- We plan to produce an ‘easy’ parallel version of established application such as Brunel, Gauss,…