17th International Conference on Computing in High Energy and Nuclear Physics (CHEP09) IOP Publishing
Journal of Physics: Conference Series **219** (2010) 042015 doi:10.1088/1742-6596/219/4/042015

User-friendly Parallelization of GAUDI Applications with Python

Pere Mato, Eoin Smith

PH Department, CERN, 1211 Geneva 23, Switzerland

E-mail: pere.mato@cern.ch

Abstract. GAUDI is a software framework in C++ used to build event data processing applications using a set of standard components with well-defined interfaces. Simulation, high-level trigger, reconstruction, and analysis programs used by several experiments are developed using GAUDI. These applications can be configured and driven by simple Python scripts. Given the fact that a considerable amount of existing software has been developed using serial methodology, and has existed in some cases for many years, implementation of parallelisation techniques at the framework level may offer a way of exploiting current multi-core technologies to maximize performance and reduce latencies without re-writing thousands/millions of lines of code. In the solution we have developed, the parallelization techniques are introduced to the high level Python scripts which configure and drive the applications, such that the core C++ application code requires no modification, and that end users need make only minimal changes to their scripts. The developed solution leverages from existing generic Python modules that support parallel processing. Naturally, the parallel version of a given program should produce results consistent with its serial execution. The evaluation of several prototypes incorporating various parallelization techniques are presented and discussed.

1. Introduction to GAUDI

GAUDI [1] is a mature software framework used by several HEP experiments, in particular LHCb and ATLAS at LHC, and which is similar in many respects to the frameworks used by other experiments. All of the data processing applications, including simulation, reconstruction, high-level trigger and analysis are based on the same framework. The GAUDI architecture considers the algorithmic part of any data processing as a set of objects that are distinct from the objects holding the data of the event. This decoupling between the objects describing the data and those implementing the algorithms allows programmers to concentrate separately on both. Applications are made by customising the framework, composing sequences of *Algorithms* and adding specific *Services* as seen in Figure 1.

1.1. Python interface

A Python interface was later added to the framework to steer and configure GAUDI applications using Python scripts. These scripts provide a convenient means for end-users to provide the configuration of several thousands of configuration parameters to steer the program. A typical script is shown in Figure 2, showing how easy is to configure the application and perform some simple data analysis. One of the advantages of using Python is that a single script can provide the complete configuration, program steering and data analysis algorithm specification.

© 2010 IOP Publishing Ltd

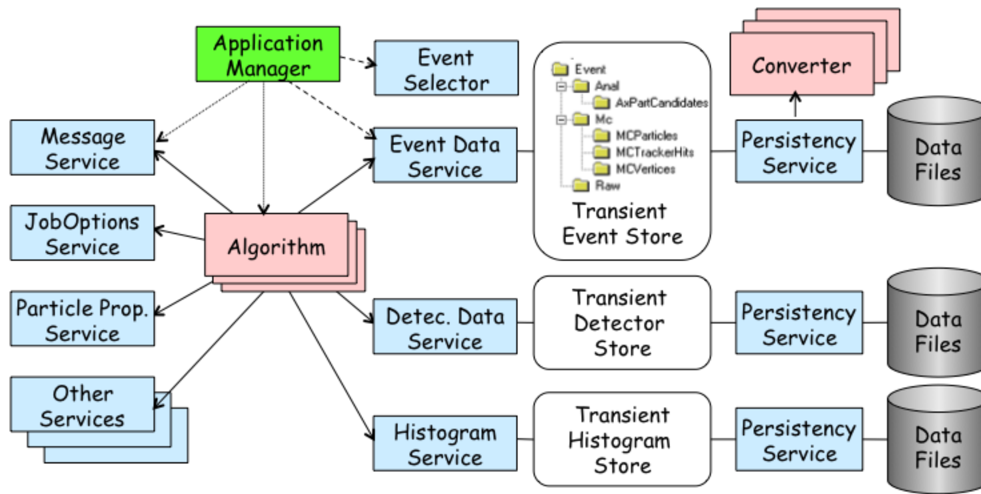

Figure 1. GAUDI object diagram

We can distinguish in the simple example the configuration phase (using some predefined standard configurations), the preparation of the analysis (booking histograms, selecting input data files, etc.), the looping over the events, and finally, presentation of the results (drawing the histograms). The main benefit of these steering scripts is that they hide the complexity of a large C++ software system, allowing end-user physicists to perform simple tasks quickly.

```
#--- Common configuration -------------------------
from Gaudi.Configuration import *
importOptions('$STDOPTS/LHCbApplication.opts')
importOptions('DoDC06selBs2Jpsi2MuMu_Phi2KK.opts')

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 and Customisations

Preparation (File Selection, Histogram booking, set up GaudiPython tools, etc.)

Loop over events Performing analysis

Present Results

Figure 2. Example of a typical GaudiPython Script. The end-user may use existing options files to configure an application run, and is then free to concentrate on the particular task they wish to achieve (in this case, filling a simple histogram from an input file)

1.2. Multi-core Architectures

The emergence of multi-core architectures has been driven by the constant need to increase and improve performance in computing, and the realisation that single-processor capabilities have

reached a plateau (around 2004) due to various limiting factors. Once the dominant determinant of processing power, we now see that considerations such as memory and power consumption have overtaken frequency as the main obstacles to be dealt with, pushing manufacturers towards alternate methods of providing better performance. As a result of this, multi-core technology stands today as the pre-eminent tool for high performance computing, from dual-core laptops to many-core workstations.

We in HEP must be able to exploit these emerging technologies to obtain the maximum performance possible. This is not a trivial task, given that the application software used runs to millions of lines of code, written over many years, with many different goals. However, we may exploit the obvious advantage of typical HEP software; event processing is embarassingly parallel. This leads us to conclude that incorporating parallelisation into the software framework would be the ideal way to adapt our software to multi-core technology without rewriting enormous amounts of serial code.

1.3. Requirements and Goals

The first and most obvious requirement is that we achieve faster turn-around times when using 4, 8, or more cores compared to the single processor scenario. We hope also to extend the parallelism effectively to 16, perhaps 24 cores, to allow for deployment on the most modern chipsets. The second consideration, of a more practical nature, is to introduce this parallelism without affecting large changes in the Python steering scripts, and without changing any of the C++ code of which GAUDI is composed.

Another requirement is that we leverage existing third-party Python modules. We do not need to re-invent anything in particular for GAUDI; if we can manage distinct worker processes, and enable inter-process communication and synchronisation with existing modules, then we should use these to build our parallel machinery.

Of course, we must also monitor the memory consumption of any parallelised application. The HEP applications used by LHCb and ATLAS are already memory-intensive. We cannot simply run N processes independently on N cores for the simple reason that the memory usage would quickly increase to unsustainable levels. We must implement memory sharing wherever possible to minimise the individual stack memory of each process. To summarize, the main requirements/goals of any parallel proposals are as follows:

- Get quicker responses when having 4-8 core machines at your disposal
- Minimal (and understandable) changes to the Python scripts driving the program. No changes in the C++ code should be needed
- Leverage from existing third-party Python modules
- Optimization of memory usage

2. Parallelisation Model

To facilitate the adaptation to different Python parallelisation modules (e.g. processing [3], parallel python [4]), each of which have different interfaces and semantics, we have introduced a very simple model for the parallel processing of the user tasks. The user only needs to provide a `Task` that implements a number of predefined methods. These methods are called at the adequate moment by the `GaudiPython.Parallel` module and hide completely the interaction with the underlying parallelisation mechanism. The results of the processing method executed in parallel are also automatically merged and returned to the main script (summing results where possible, or appending to Python lists). In this way, the user focuses on his algorithm. Results can be any pickable python objects. This includes any C++ object for which we have

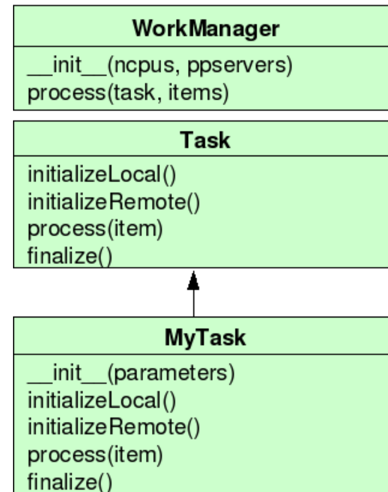

Figure 3. Simple parallelisation model. The user tasks inherits from *Task* and implements a number of methods.

a Reflex [2] dictionary. In this way the end-user does not need to modify any scripts when this new functionality is included.

The same example from Figure 2, adapted to the model is shown in Figure 4. As can be inferred, it is not a very difficult change and conceptually very simple. The net result for the user is that now the script is able to exploit all the available cores.

```
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 initializeRemote(self):
    importOptions('$STDOPTS/LHCbApplication.py')
    importOptions('$BS2JPSIPHIROOT/options/DoDC06selBs2Jpsi2MuMu_Phi2KK.py')
  def process(self, file):
    appMgr = AppMgr() ; evt = appMgr.evtsvc()
    appMgr.evtsel().open([file])
    while 0 < 1:
      appMgr.run(1)
      if evt['Rec/Header'] == None : break
      cont = evt['Phys/DC06selBs2Jpsi2MuMu_Phi2KK/Particles']
      if cont:
        for b in cont : self.output['h_bmass'].Fill(b.momentum().mass())
  def finalize(self):
    self.output['h_bmass'].Draw()
    print 'Bs found: ', self

from inputdata import bs2jpsiphimm_files as files
if __name__ == '__main__' :
  task = MyTask()
  wmgr = WorkManager()
  wmgr.process( task, files[:50])
```

Figure 4. Python steering script from Fig 2 adapted to use `GaudiPython.Parallel` module. By filling in the methods provided by the `Task` class, the user may then submit this custom `Task` to the `WorkManager`, which implements parallelisation using cluster or multi-core methods, transparent to the user.

3. Performance Measurements

We have made some initial performance measurements based on variations on the simple scripts shown in Figure 2 and Figure 4. Using 50 input files, the `Task` was distributed among n worker processes. Two use-cases were examined; local and remote files, to investigate the effect of I/O bound systems on parallelisation. The results are seen in Figures 5-6

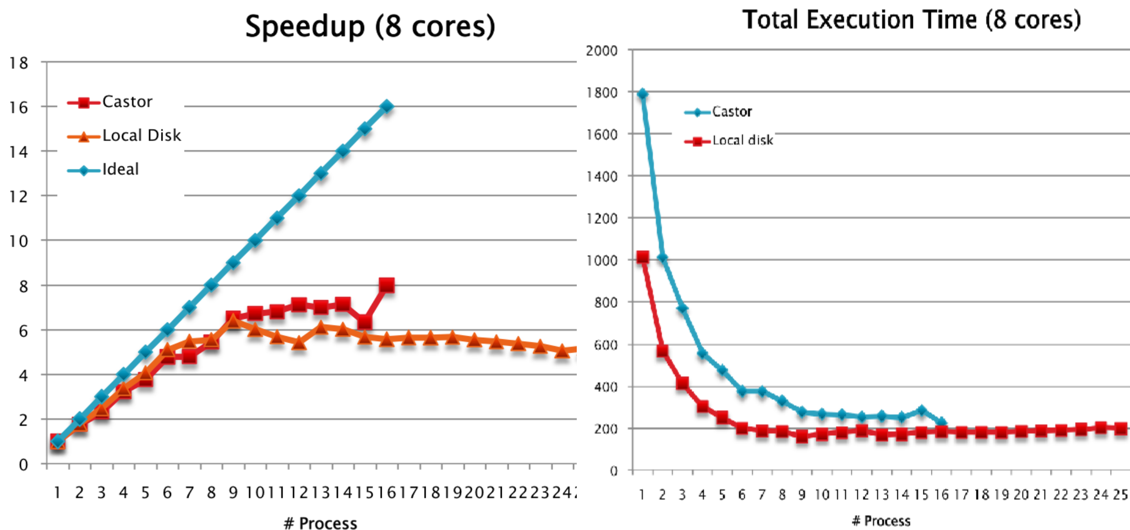

Figure 5. Speedup: histogram filling, 50 input files (local/remote). Both cases show sub-linear speedup as the number of processors increases.

Figure 6. Total Execution time of the test run in Figure 5. Remote file access protocols delay access and cause cpu utilisation inefficiencies.

We see from Figure 5 that although the speedup begins to level off with more than ten processes, speedup of 7 is possible with 9 processes. Since todays chipsets offer, in general 4-8 cores, this offers suitable speedup with little inconvenience to the end-user.

4. Memory Usage

As mentioned in Section1.3, one of the priorities in developing a parallelisation scheme for Gaudi is to minimise the size of the memory footprint. Wherever possible, memory should be shared to avoid needless duplication of constant data such as detector geometry and initialisation tables. To this end, we monitor the memory consumption using the `smaps` script, written in Python to digest the data contained in `/proc/PID/smaps`. An example is shown in Figure 7.

```
3a5c100000-3a5c115000 r-xp 00000000 08:03 621533  /lib64/ld-2.3.4.so
Size:                  84 kB
Rss:                   76 kB
Pss:                    1 kB
Shared_Clean:          76 kB
Shared_Dirty:           0 kB
Private_Clean:          0 kB
Private_Dirty:          0 kB
Referenced:            76 kB
```

Figure 7. Example of data contained in `/proc/PID/smaps`

The Copy-on-Write technique offered by the operating system ensures that initially, all data is shared, and subsequent modifications results in a local copy of the memory. Therefore, if we

perform as much initialisation as possible on the parent process before forking, then we maximise the amount of shared memory.

```
from Gaudi.Configuration import *
from GaudiPython.Parallel import Task, WorkManager
from GaudiPython import AppMgr, PyAlgorithm

class Brunel(Task):
  def initializeLocal(self): pass

  def initializeRemote(self):
    importOptions('$BRUNELROOT/
                   options/Brunel-Default.py')
    importOptions('$BRUNELROOT/
                  options/DC06-Files.py')
    ApplicationMgr( OutputLevel = ERROR,
                   AppName = 'PyBrunel')
    appMgr = AppMgr()
    appMgr.initialize()
    ...
```

```
from Gaudi.Configuration import *
from GaudiPython.Parallel import Task, WorkManager
from GaudiPython import AppMgr, PyAlgorithm

importOptions('$BRUNELROOT/
              options/Brunel-Default.py')
importOptions('$BRUNELROOT/
              options/DC06-Files.py')
ApplicationMgr( OutputLevel = ERROR,
               AppName = 'PyBrunel')
appMgr = AppMgr()
appMgr.initialize()

class Brunel(Task):
  def initializeLocal(self): pass

  def initializeRemote(self): pass
```

Figure 8. Two versions of the same script. In the first one, initialisation is performed by all sub-processes. In the second, the parent process performs all initialisation before forking, resulting in more efficient memory utilisation.

We may observe (Figure 8) how implementing more memory-efficient shared initialisation requires no modifications to the code beyond moving the configuration lines outside of the Task class received by the worker sub-processes, and having the parent process perform all initialisation. If any changes are to be made to the configuration, Copy-on-Write ensures that memory is only copied where necessary.

The effect on memory consumption may be seen in Table 4

Table 4: Memory Usage by different initialisation sequences: LHCb Reconstruction program (Brunel)

Scheme	ncpus	Code		Data		Sum	Total	$\frac{Total}{ncpus}$
		Private	Shared	Private	Shared			
Single	1	157	5	336	0	498	498	498
Single	2	4	167	339	0	510	850	425
Single	4	0	171	321	0	492	1462	365
Fork^a	4	0	149†	313	16	478	1454	363
Fork^b	2	0	116†	180	162	458	742	371
Fork^b	4	0	117†	178	158	453	1105	276
Fork^c	4	0	86†	118	196	400	852	213
Fork^{c,d}	8	0	86†	118	196	400	1320	165

Legend: a : No initialisation on parent process b : All services initialised on parent. c :All services initialised on parent, and one event run on parent d : Estimated values only.
† : Part of the code only mapped in parent process

We may see from Table 4 that having each sub-process perform initialisation results in much greater memory consumption (Fork^a; 1454Mb Total, 363Mb/processor for 4 processors). Allowing the parent process to initialise before forking (Fork^b on 4 processors) results in a reduction in total memory consumption of over 300Mb, or almost 100Mb per processor. Further, as processing of the first event typically involves some extra initialisation, such as the loading of tables or other services, if we perform all initialisation *and* processing of the first event before forking (Fork^c), we reduce the total memory consumption by 600Mb (over 100Mb per

processor), or just over 40%. One concern may be that each event changes the conditions slightly, resulting in a reduction of shared memory as the run progresses. However, according to more recent measurements [5], shared memory does not degrade as the number of events processed increases.

5. Model fails for 'Event' output

In the case where the application generates a lot of data for each item to be processed, then the current model fails, as final merging is done only when all events have been processed. This may produce a lot of data in some cases, making the temporary storage of results in memory insufficient. This behaviour is typical of event processing applications such as the simulation or reconstruction programs. These programs compute new event data from old data by event by event. If the processing is done over thousands of events is obvious that we will have a memory problem. To overcome this , we have developed a new design of the parallel program. To facilitate an event-distribution model, we consider an adapted version of the tradition "Scatter-Gather" scenario. In the proposed model, we assign all I/O operations to two processes, denoted "Reader" and "Writer". The idea is that the reader will distribute events to a pool of workers. Each worker will process one event at a time, sending the processed events to the writer for output. An outline of the proposed scheme is shown in Figure 9

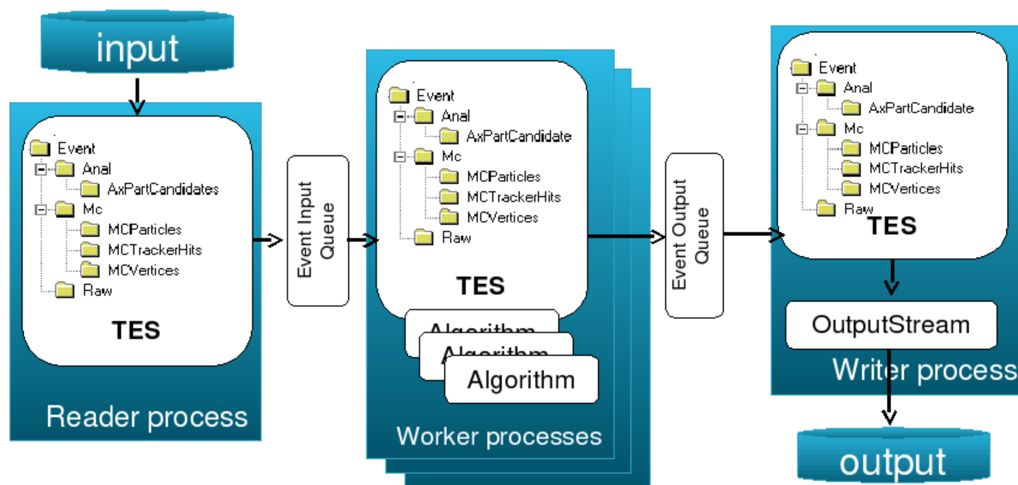

Figure 9. Simple outline of an event-based parallelisation model. No I/O is performed by the workers; there is one reader for loading input data, and a writer for writing the output file. All algorithms are applied by the worker.

5.1. Implementing the proposed model

To implement the model proposed and shown in Figure 9, we need to be able to create and manipulate subprocesses which will perform the functions of I/O and event processing. We also need a method of transferring event data between processes; from the reader to the writer via the pool of workers. As mentioned earlier, we intend to leverage existing third party Python modules where possible, and in this case, the `processing` module is a good fit.

For event distribution, it is necessary to have some means of moving the event data between processes. The GAUDI TES is not compatible with existing Python methods for inter-process communication, as it is not directly a pickable object. To overcome this, we have created a

`TESSerializer` mechanism to serialize user-specified parts of the TES for transferring portions of events and their meta-data between processes.

6. Outlook and further work

Future work will focus on event-parallelism, and producing a parallelised framework which produces identical results to traditional serial execution. This should be done in a way that available parallel resources are utilised automatically, with minimal user specification (perhaps a flag in the steering script). Parallelisation should offer scalable speedup on single input files, not just lists of input files. We hope to use the `TESSerialiser` in conjunction with fast and efficient inter-process communication to build a parallel system where one reader and one writer can support an arbitrary number of workers. This ought to be possible, as processing an event generally takes longer than writing an event, so the speedup offered should conform to Formula 1 where $T_{process}$ is the time take to process one event, and T_{write} is the time taken to write one event.

$$N_{procs} = \frac{T_{process}}{T_{write}} \tag{1}$$

There are many optimisations can be examined. We intend to examine all possible ways to minimise the amount of data to be transferred. The question of memory consumption shall be continuously examined, using tools such as the `smaps` script. Also part of the future work is fully incorporating the parallelisation into GAUDI, which means extensive testing based on all GAUDI applications (simulation,digitisation,reconstruction and analysis), as well as producing and maintaining a release version of transparent parallelisation software.

7. Conclusions

The possibilities of multi-core computing must be exploited for two reasons; multi-core technology has become the dominant technology for high-performance computing, and many of the typical HEP applications are exceptionally suitable for parallel computing. Initial tests indicate that existing third party tools such as the Python module `processing` are sufficient and reliable in providing the basic methods for inter-process communication and sub-process management. We see the possibility of introducing a Python parallel module to the GaudiPython package to offer transparent parallelisation, allowing enormous amounts of existing serial code to leverage the performance increases offered by modern multi-core hardware. We have developed a tool `TESSerializer` to allow transfer of the GAUDI TES between processes, enabling the building of a prototype where I/O is de-coupled from execution, allowing an arbitrary number of workers to receive events for processing. We intend to adapt this prototype to suit existing HEP applications, while also optimising memory consumption by using shared memory where possible. As this project is part of the recently started CERN R&D, we also intend to continue investigation and testing on existing and emerging parallel technology.

References

[1] G. Barrand *et al.*, "GAUDI - A software architecture and framework for building HEP data processing applications," Comput. Phys. Commun. **140** (2001) 45.
[2] S. Roiser and P. Mato, "The SEAL C++ reflection system," *In *Interlaken 2004, Computing in high energy physics and nuclear physics* 437-440*
[3] `http://pypi.python.org/pypi/processing`
[4] `http://www.parallelpython.com/`
[5] private communication: Jeff Arnold/CERN openlab