Performance Optimization

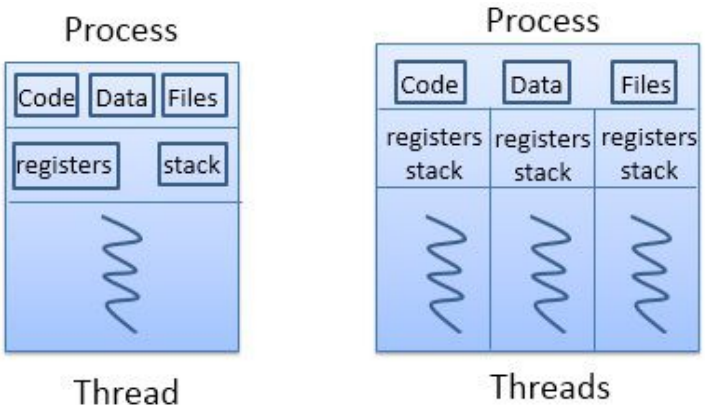
Justin Anguiano

August 20, 2019

Goal — Improve Physics analysis performance

Use different C/Python approaches to obtain a low programming overhead but high performance

Today: a look at multithreading and multiprocessing in C++ ROOT



Multithreading

Multiprocessing-

- adds CPUs to increase performance
- processes run concurrently
- creating/managing processes is resource intensive
- processes do not share memory

Multithreading-

- adds threads to a single CPU to increase performance
- multithreading is economic for the system
- threads share memory

There are parallelization tools to exploit both of these approaches built into root!

Definitions:

thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler
The number of threads possible depend on the number of cores in the CPU

process is the instance of a computer program that is being executed by one or many threads

Test 1: Local multi-threading

Task to be optimized: Read File(s) containing a TTree, produce histograms from elements of the tree(s), write histograms to a TFile

Three histograms are produced: TH1D: track p_T weighted by $1/p_T$, TH1D: track p_z, TH2D, track p_x vs p_y

First attempt using ROOT6 parallelization on local machine (laptop) methods:

- Sequential run over a test file with TTreeReader (Compiled and Interpreted)
- Parallelized run over a test file with TTreeReader (Compiled and Interpreted)

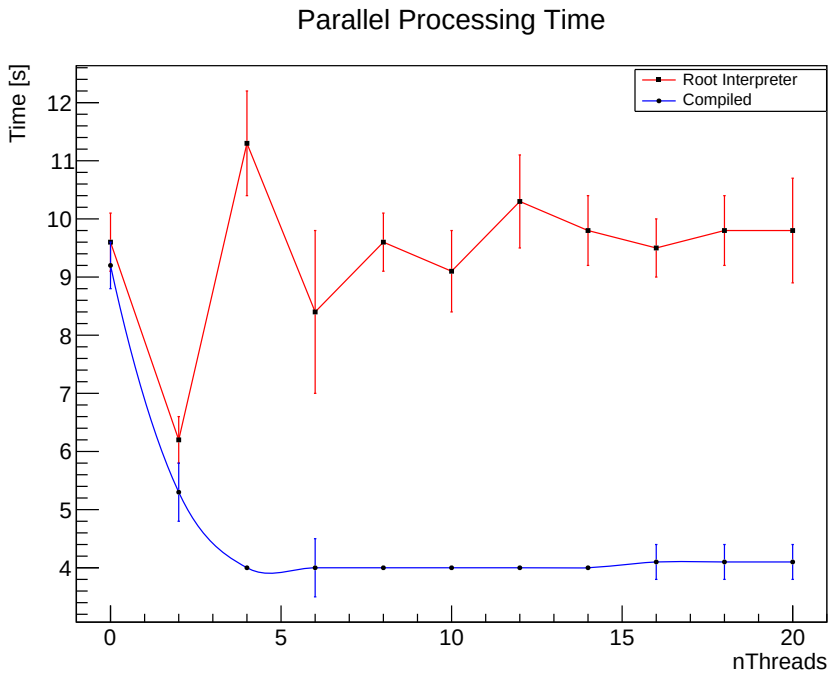
Test File Details

- code adapted from example: `https://root.cern.ch/doc/v612/imt101__parTreeProcessing_8C.html`
- uses fake event file containing 48,000 events; each with some number of tracks per event
- total tracks overall $\approx 2.4e+7$

Test 1: Local multi-threading

First Test Results:

- Time is Mean time $\pm$ stdev over 10 trials
- no guarantee of system releasing requested resources (nthreads) so perform multiple trials
- data point at nThreads $= 0$ is the basic sequential program
- high threadcount performance possibly bottlenecked by system?
- Takeaway: Compiling is important! $\approx$Factor of 2 improvement



Test 2: UNL multi-threading

Second attempt using ROOT6 parallelization versus classic(Make Class) sequential program (my current approach) on t3.unl.edu
Methods:

- Sequential run over multiple test files with TTreeReader (Compiled Only)
- Parallelized run over multiple test files with TTreeReader (Compiled Only)
- Sequential run over multiple test files with MakeClass (Compiled and Interpreted)

Test File set Details:

- using 9 files from Dataset: `/SingleMuon/Run2018D-PromptReco-v2/AOD`
  - Physics group: NoGroup Creation time: 2018-08-01 13:16:41 Status: VALID Type: data Dataset size: 150070502441346 (150.1TB) Number of blocks: 267 Number of events: 511823047 Number of files: 45330
- 1,576,737 events each with at least 1 conversion per event
- total tracks overall (exactly 2 per conversion) = 14,190,956

Test 2: UNL multi-threading

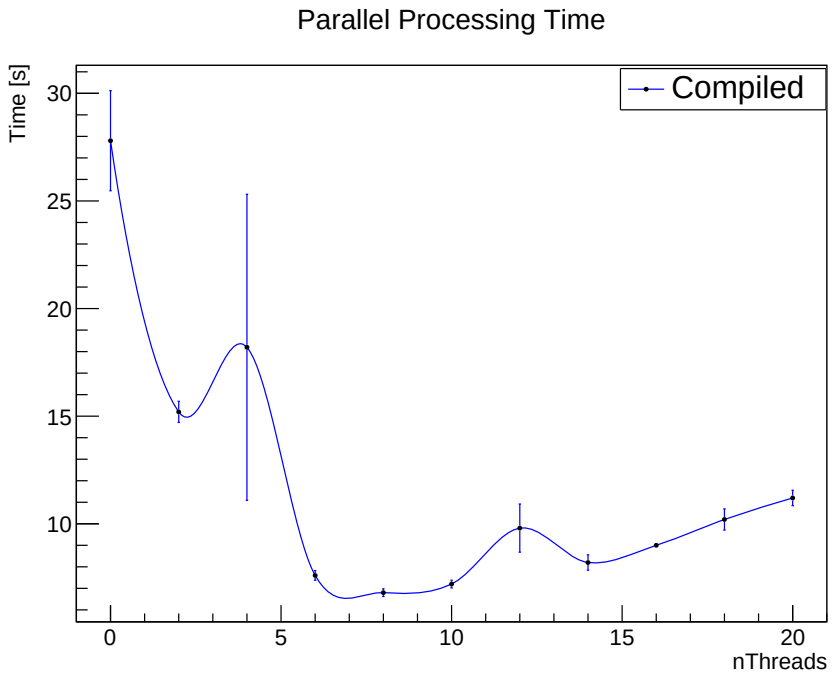
File Sequence Details:

- `Run2018_100.root`
  - contains 193388 events
  - contains 1547322 unique conversion tracks
- `Run2018_110.root`
  - contains 140502 events
  - contains 950444 unique conversion tracks
- `Run2018_120.root`
  - contains 99085 events
  - contains 602330
- `Run2018_130.root`
  - contains 62289 events
  - contains 347156 unique conversion tracks
- `Run2018_141.root`
  - contains 218339 events
  - contains 1958272 unique conversion tracks
- `Run2018_155.root`
  - contains 269483 events
  - contains 2968340 unique conversion tracks
- `Run2018_166.root`
  - contains 172409 events
  - contains 1299172 unique conversion tracks
- `Run2018_176.root`
  - contains 127117 events
  - contains 832024 unique conversion tracks
- `Run2018_193.root`
  - contains 294125 events
  - contains 3685896 unique conversion tracks

First Cluster Test Results:
(Using full 9 file dataset)

- Time is Mean time $\pm$ $s/\sqrt{N}$ over 5 trials
- data point at nThreads $= 0$ is the basic sequential program
- high threadcount again not optimal, chokes up the system
- Classic sequential approaches using

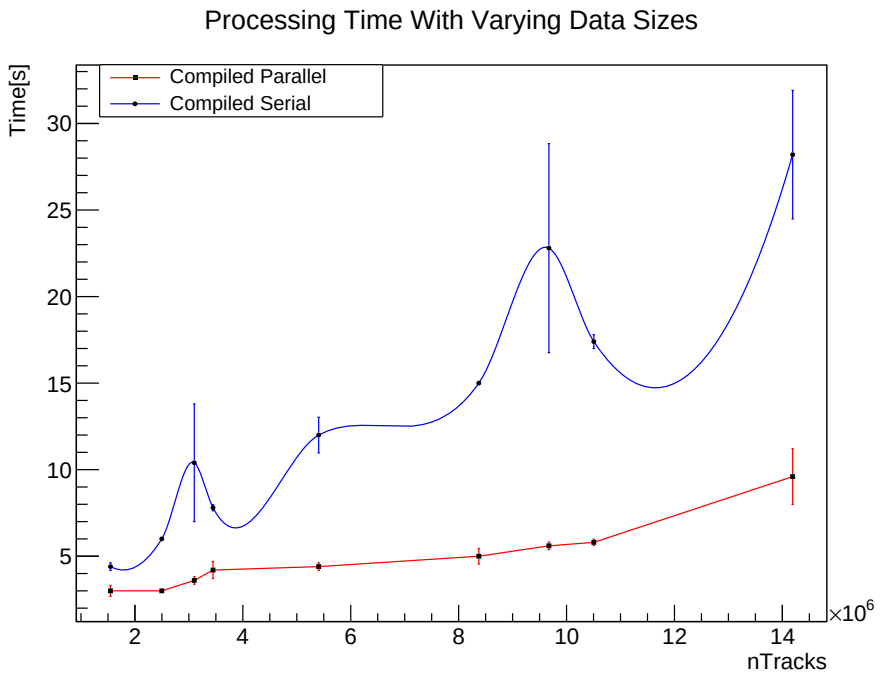
  MakeClass (Compiled and Interpreted)–
  - 265.4 ± 9.2 [s] (Interpreted)
  - 171.0 ± 10.6 [s] (Compiled)
- ROOT6 mutlithreading is very good!



Test 2: UNL multi-threading

Second Cluster Test:The impact of data size on performance

- Time is Mean time $\pm\ s/\sqrt{N}$ over 5 trials
- nTracks is the accumulated number of tracks looped over for a given number of files that are run in a preserved order
- Parallel is run with the previously optimal number of threads (8)
- Parrellization is faster and more consistent at any size of dataset
- As data gets larger the parallel benefit becomes more significant



Test 3: UNL multi-processing

Two "new" tools:
Parallel ROOT Facility (PROOF) and **Selectors**

PROOF and Selector are designed be be used together
PROOF is a toolset used to manage/delegate processes between nodes on a cluster
Selector is the ROOT6 version of a MakeClass i.e. TTreeReaderValues/Arrays

Using them is easy:

- write a simple analysis in Selector
- load all input files into a TChain
- create a PROOF instance
- tchain.Process("selector.C")

Test 3: UNL multi-procesing

Test 3: compare multiprocessing and multithreading at UNL
- use the same 9 files with the same task
- also try a large 918 file (Single Muon) test case with the same task

the large dataset contains 146,607,893 events

Method	numFiles	nTrials	Time (s)
multiThread	9	5	6.8 ± 0.2
multiProcess	9	3	92.3 ± 0.9
multiThread	918	1	26m 4s
multiProcess	918	1	126m 56s

Multithreading is still significantly faster!!
Process resource and management overhead is high
Possible to combine Multiprocessing + Multithreading in ROOT??
In principle yes..

Application to SUSY Analysis

New tool availble at github : githublink.com
Basic Analysis template – good for any Tree reading + Histogram producing analysis
Modular tools assembled to automatically parse Trees and perform multithreading $\rightarrow$ just make histograms!

How it works:

- uses a Python script for input interface
  - just feed in a list of signal or background rootfiles with the treename
- uses a TSelector to access all possible TTreeReaderValue branches in analysis
  - easily automatically generated with just tree.MakeSelector("myselector")
- histset.c is the meat of the analysis, contains threaded histograms and physics analysis code
  - this is the only place where code needs to be modified
- the custom class ParTreeProcessing.c manages the selector/multithreading/histograms

Application to SUSY Analysis

Test 4: Multithreaded SUSY Analysis Running over all large backgrounds and two signal trees

Background: (all HT binned)
`DYJetsToLL_HT.list`
`QCD_HT.list`
`TTJets_HT.list`
`WJetsToLNu_HT.list`
`ZJetsToNuNu_HT.list`

Signal:
`SMS_TChiWZ_ZToLL.list`
`SMS_300_75` and `SMS_175_135`

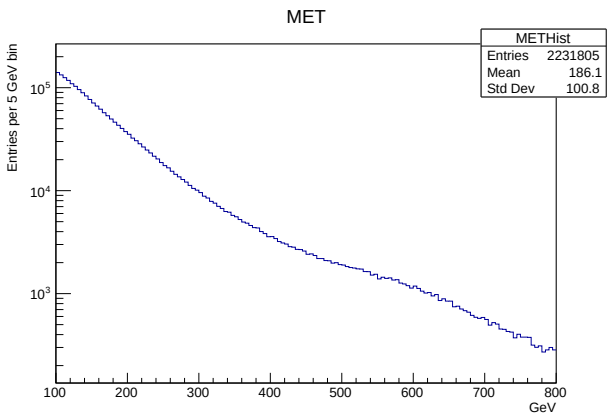
SUSY Results

```
    PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 941900 janguian  20   0 2438m 2.0g  71m R 794.9  2.9   5:18.92 compiledthreads
 936317 crogan    20   0  754m 213m  48m R 99.6  0.3   7:48.96 MakeReducedNtup
 928604 crogan    20   0  754m 193m 6248 R 99.0  0.3  15:39.93 MakeReducedNtup
 927931 crogan    20   0  754m 193m 4472 R 81.7  0.3  15:56.71 MakeReducedNtup
 913844 erichjs   20   0 1219m 744m  91m S 45.9  1.0  19:42.86 python
 913694 erichjs   20   0 1223m 748m  91m S 45.6  1.0  19:12.97 python
```

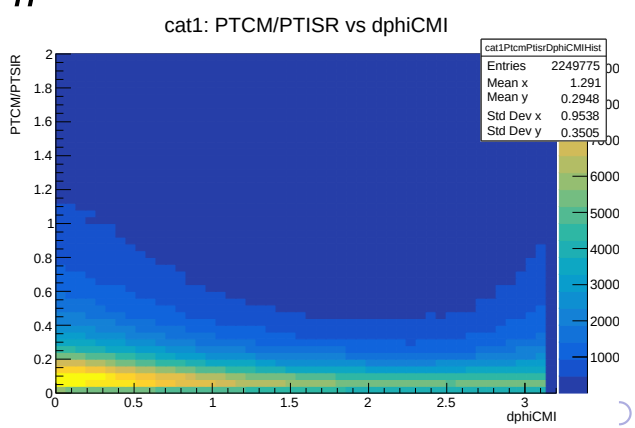
```
root [1] _file0->ls();
TFile**         susyHists.root
 TFile*         susyHists.root
  KEY: TH1D     DYMETHist;1     object title
  KEY: TH1D     DYcat0NjetSHist;1       object title
  KEY: TH1D     DYcat1NjetSHist;1       object title
  KEY: TH2D     DYcat0PtcmPtisrDphiCMIHist;1    object title
  KEY: TH2D     DYcat1PtcmPtisrDphiCMIHist;1    object title
  KEY: TH1D     WJETMETHist;1   object title
  KEY: TH1D     WJETcat0NjetSHist;1     object title
  KEY: TH1D     WJETcat1NjetSHist;1     object title
  KEY: TH2D     WJETcat0PtcmPtisrDphiCMIHist;1  object title
  KEY: TH2D     WJETcat1PtcmPtisrDphiCMIHist;1  object title
  KEY: TH1D     QCDMETHist;1    object title
  KEY: TH1D     QCDcat0NjetSHist;1      object title
  KEY: TH1D     QCDcat1NjetSHist;1      object title
  KEY: TH2D     QCDcat0PtcmPtisrDphiCMIHist;1   object title
  KEY: TH2D     QCDcat1PtcmPtisrDphiCMIHist;1   object title
  KEY: TH1D     TTJMETHist;1    object title
  KEY: TH1D     TTJcat0NjetSHist;1      object title
  KEY: TH1D     TTJcat1NjetSHist;1      object title
  KEY: TH2D     TTJcat0PtcmPtisrDphiCMIHist;1   object title
  KEY: TH2D     TTJcat1PtcmPtisrDphiCMIHist;1   object title
  KEY: TH1D     ZNUNUMETHist;1  object title
  KEY: TH1D     ZNUNUcat0NjetSHist;1    object title
  KEY: TH1D     ZNUNUcat1NjetSHist;1    object title
  KEY: TH2D     ZNUNUcat0PtcmPtisrDphiCMIHist;1 object title
  KEY: TH2D     ZNUNUcat1PtcmPtisrDphiCMIHist;1 object title
  KEY: TH1D     WZ_300_75METHist;1      object title
  KEY: TH1D     WZ_300_75cat0NjetSHist;1        object title
  KEY: TH1D     WZ_300_75cat1NjetSHist;1        object title
  KEY: TH2D     WZ_300_75cat0PtcmPtisrDphiCMIHist;1     object title
  KEY: TH2D     WZ_300_75cat1PtcmPtisrDphiCMIHist;1     object title
  KEY: TH1D     WZ_175_135METHist;1     object title
  KEY: TH1D     WZ_175_135cat0NjetSHist;1       object title
  KEY: TH1D     WZ_175_135cat1NjetSHist;1       object title
  KEY: TH2D     WZ_175_135cat0PtcmPtisrDphiCMIHist;1    object title
  KEY: TH2D     WZ_175_135cat1PtcmPtisrDphiCMIHist;1    object title
```

plots produced: MET, Njet_ S , PTCM/PTISR vs dPhiCMI

$Z \to \nu\nu$



$Z \to ll$



Final Runtime: Real time– 234.2 ± 10.2 s $\approx (4m)$ System time: 61.7 ± 1.6 s

Conclusion

Multithreading is very very fast! – we need to use this –

My analysis template can also be adapted to other tasks i.e. making reduced Ntuples

Next up: Vectorization, Scientific Python - uproot etc.