# Benchmarking CORSIKA 8

**Nikolaos Karastathis, Pranav Sampathkumar and Maximilian Reininghaus**

**for the CORSIKA 8 collaboration**


# Outline

- 10 vertical hadronic showers
- 5 45 degrees em showers
- Runtime vs energy cut for hadronic showers
- Profiling

# Workstation specifications:

- CPU: AMD Ryzen Threadripper 2970WX 24-Core Processor
  Base clock speed: 3GHz
  Max clock speed: 4.2GHz
  L3 cache: 64 MB
- RAM memory: 64 GB

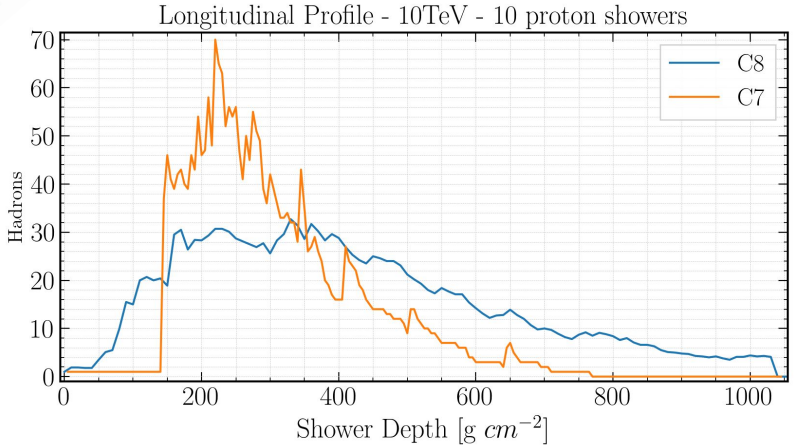
# Run specifications:

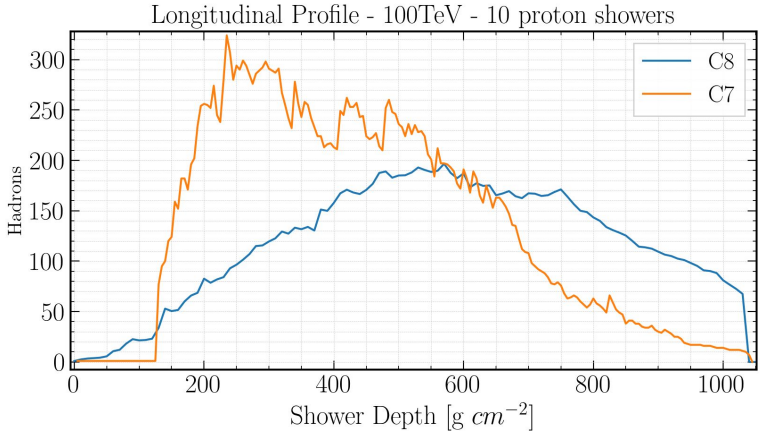
- C7: FFLAGS=-O2 CFLAGS=-O2 ./coconut
  – QGSJETII-04, URQMD 1.3cr
- C8: vertical_EAS example
  – logging set to WARN
  – CUTS were set to appropriate values in order to avoid crushes in PROPOSAL
  – Trackwriter is disabled

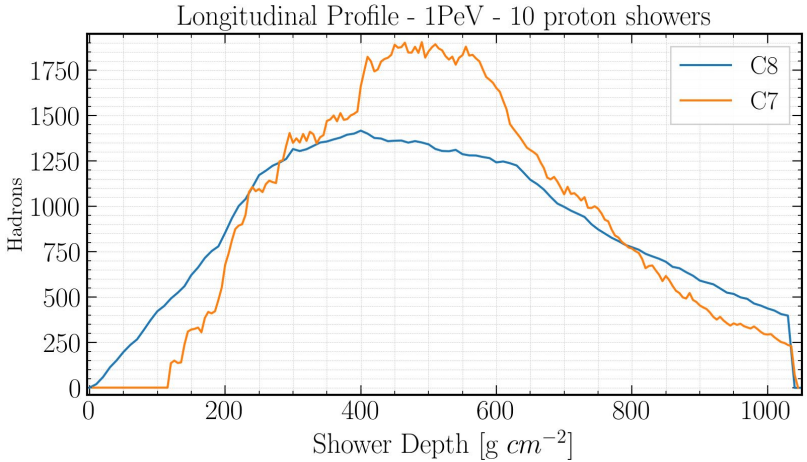
# Hadronic and EM showers were produced:

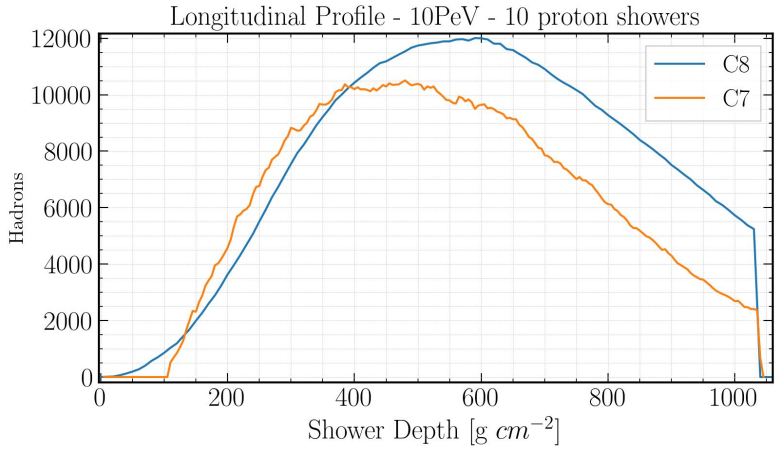
- branch: 502-examples-need-some-polishing
- commit: 94875ed461a9de73d06881ea0d83941833c99fb4
- random seeds were used
- examples: vertical_EAS.cpp, em_shower.cpp

# Hadronic showers



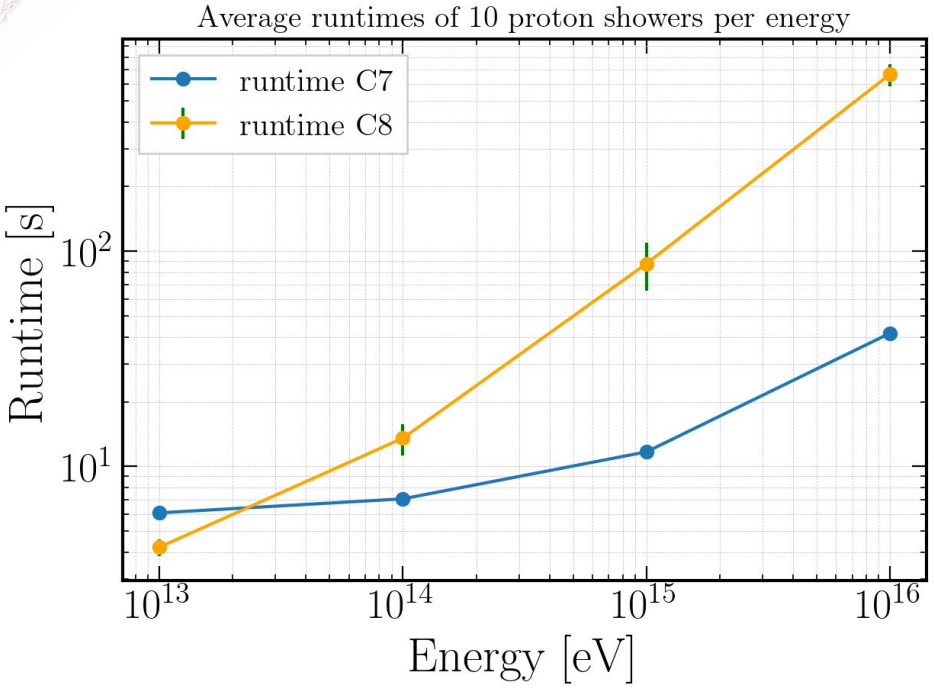



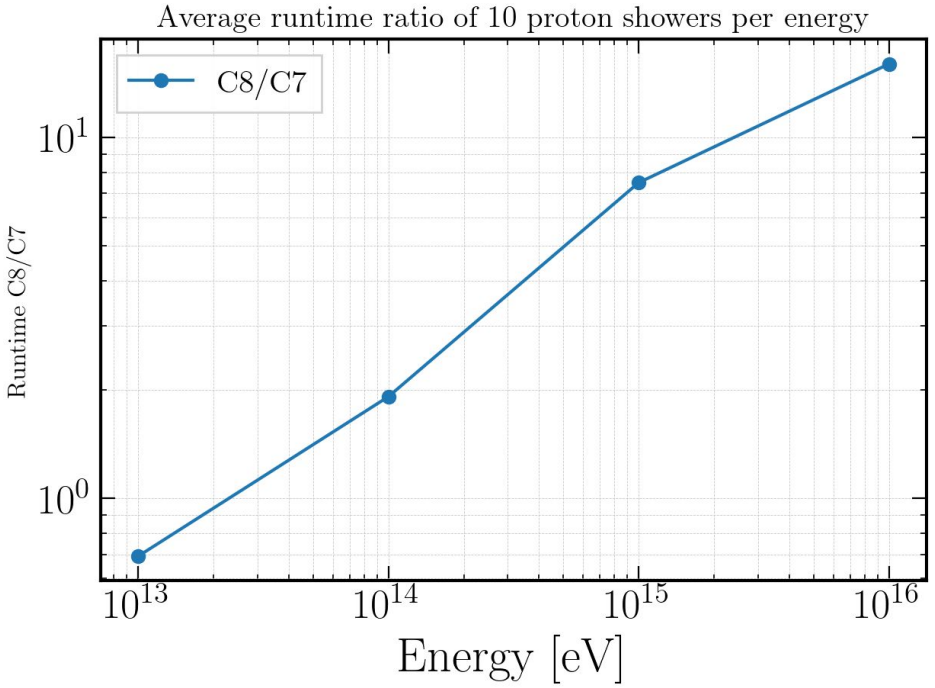



# Hadronic showers

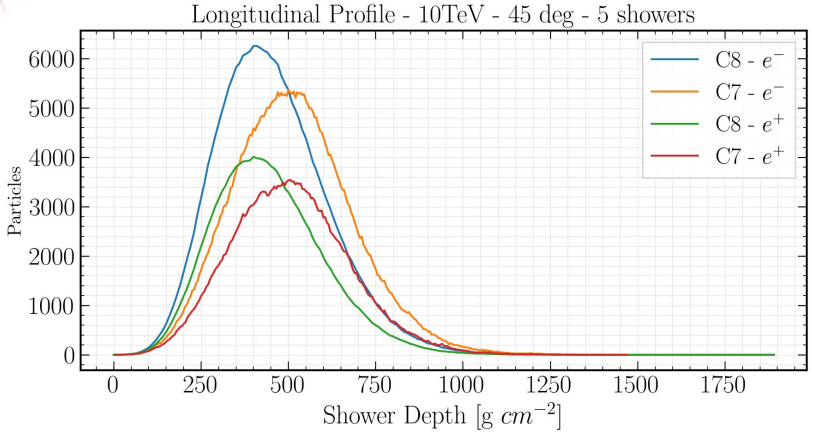


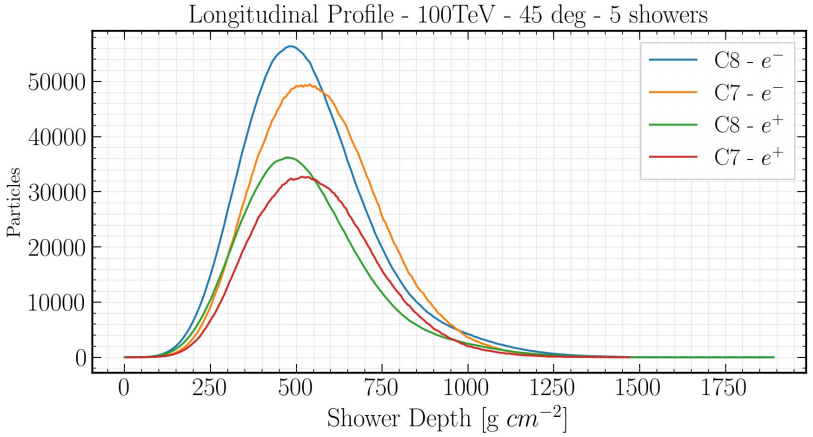


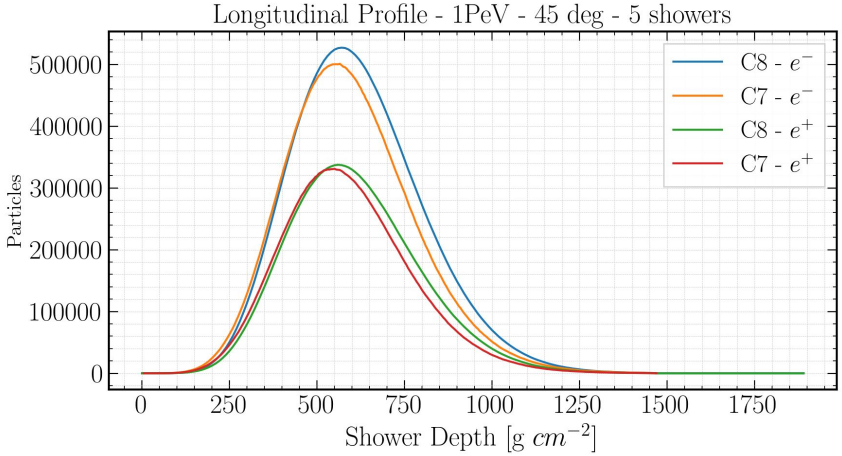



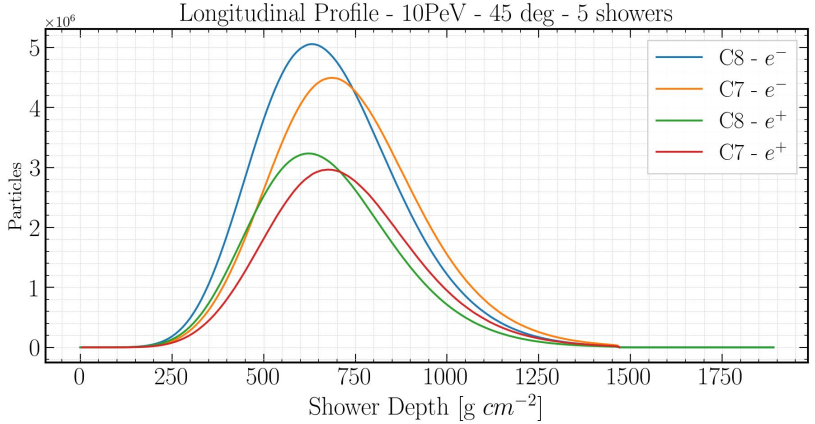
# EM showers




Longitudinal Profile - 10TeV - 45 deg - 5 showers
Particles
Shower Depth [g $cm^{-2}$]
C8 - $e^-$
C7 - $e^-$
C8 - $e^+$
C7 - $e^+$

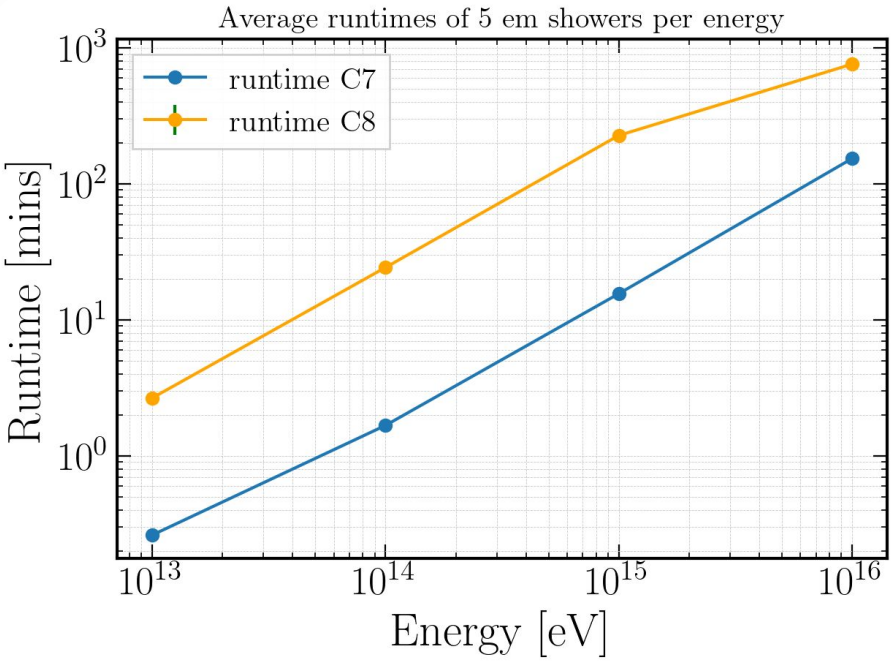

Longitudinal Profile - 100TeV - 45 deg - 5 showers
Particles
Shower Depth [g $cm^{-2}$]
C8 - $e^-$
C7 - $e^-$
C8 - $e^+$
C7 - $e^+$

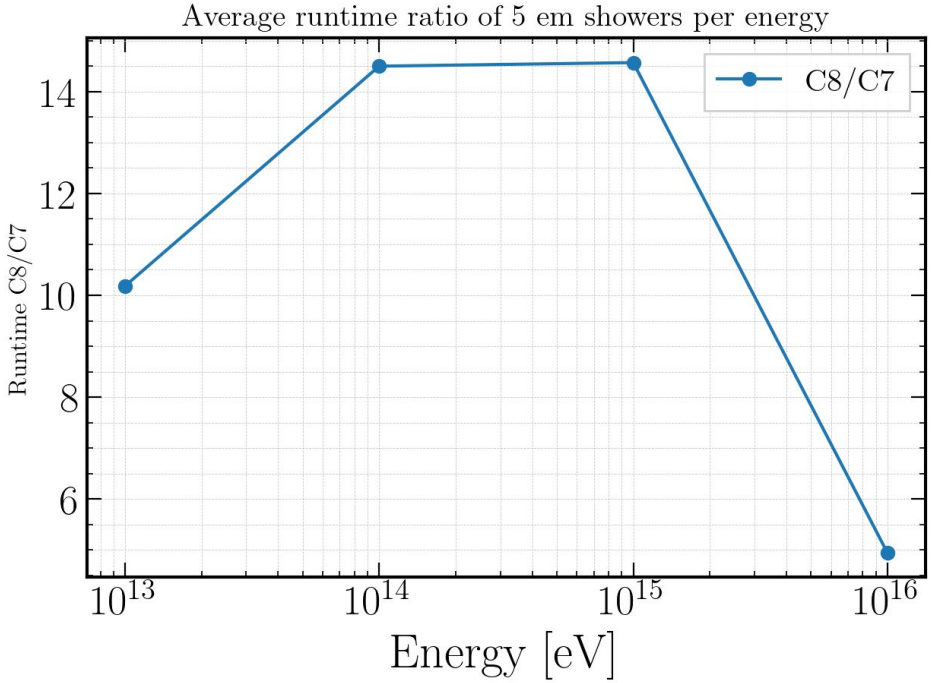

Longitudinal Profile - 1PeV - 45 deg - 5 showers
Particles
Shower Depth [g $cm^{-2}$]
C8 - $e^-$
C7 - $e^-$
C8 - $e^+$
C7 - $e^+$


Longitudinal Profile - 10PeV - 45 deg - 5 showers
$\times 10^6$
Particles
Shower Depth [g $cm^{-2}$]
C8 - $e^-$
C7 - $e^-$
C8 - $e^+$
C7 - $e^+$

# EM showers




Average runtimes of 5 em showers per energy
runtime C7
runtime C8
Runtime [mins]
$10^3$
$10^2$
$10^1$
$10^0$
$10^{13}$
$10^{14}$
$10^{15}$
$10^{16}$
Energy [eV]

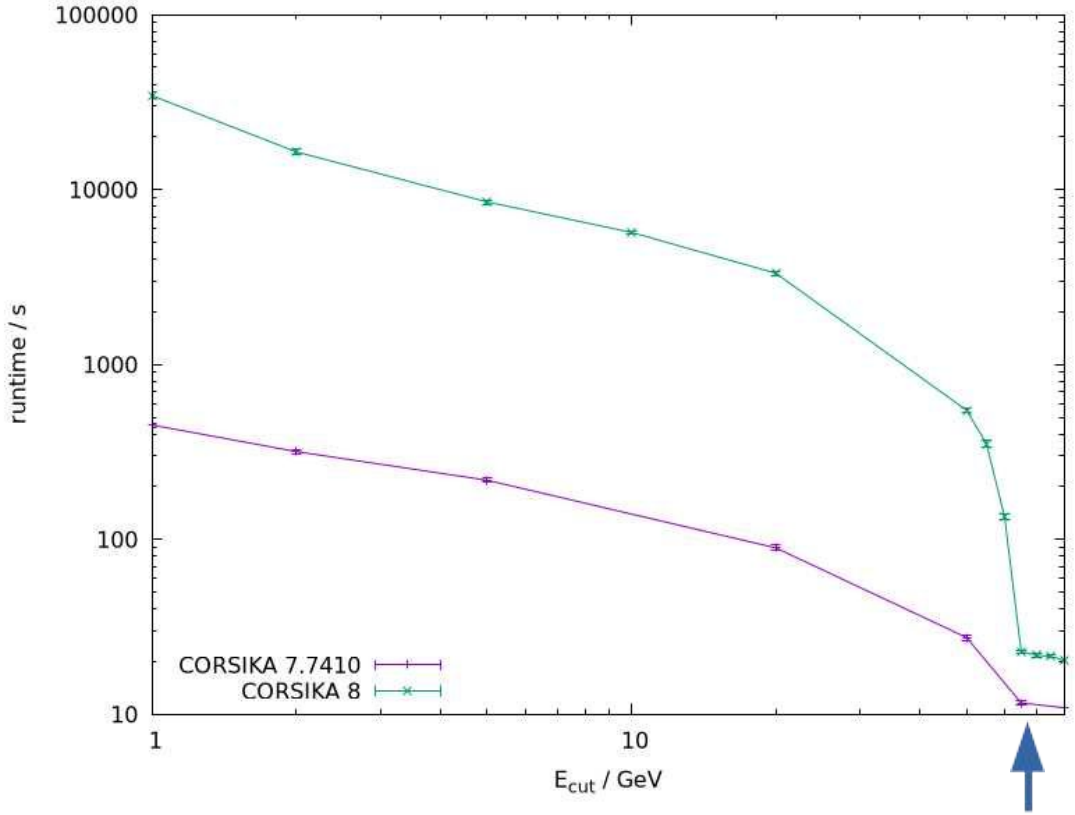

Average runtime ratio of 5 em showers per energy
C8/C7
Runtime C8/C7
14
12
10
8
6
$10^{13}$
$10^{14}$
$10^{15}$
$10^{16}$
Energy [eV]

# Runtime vs energy cut - 40 hadronic showers



primary: p 1017 eV, vertical
C8: straight tracking, BetheBlochPDG
C7: ELMFLG = 0, compiled with -O2

- These showers were run by Max in his station




LE/HE transition

# Profiling



## Corsika details

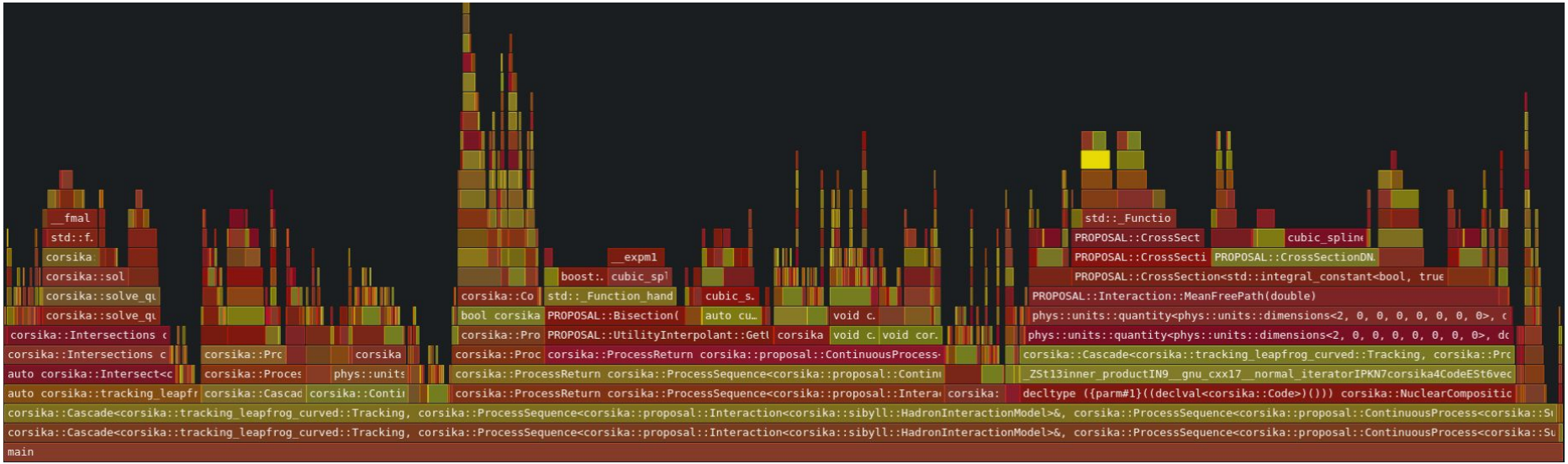
- branch: 502-examples-need-some-polishing
- commit: 94875e
- example: em_shower.cpp
- 100 TeV, Vertical Shower, Trackwriter disabled

## Environment (Pranav's computer)

- Compilation tools: gcc=11.3.1, cmake=3.22.2
- OS: Fedora 35
- Processor: AMD Ryzen 7 PRO 3700 8-Core Processor (3.6GHz - 4.4GHz)
- Performance Analysis Tools: Valgrind=3.19, perf=5.18.4, hotspot=1.3.0, kcachegrind=21.12.2

# Flame Graph

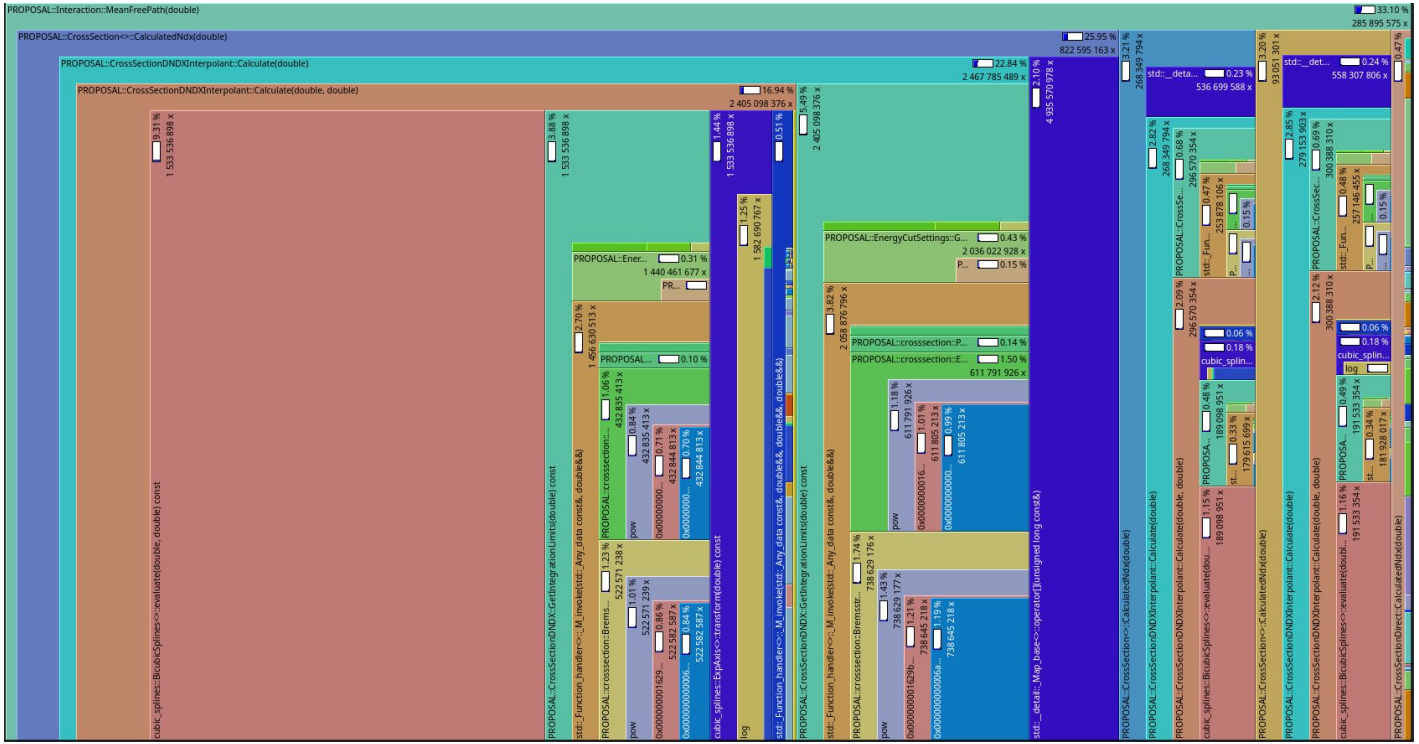





Flame Graph for CORSIKA8, this flamegraph helps us spot a few important hotspot functions, which we can analyse further, such as GetUpperLimit and MeanFreePath from PROPOSAL.

# Callcache information

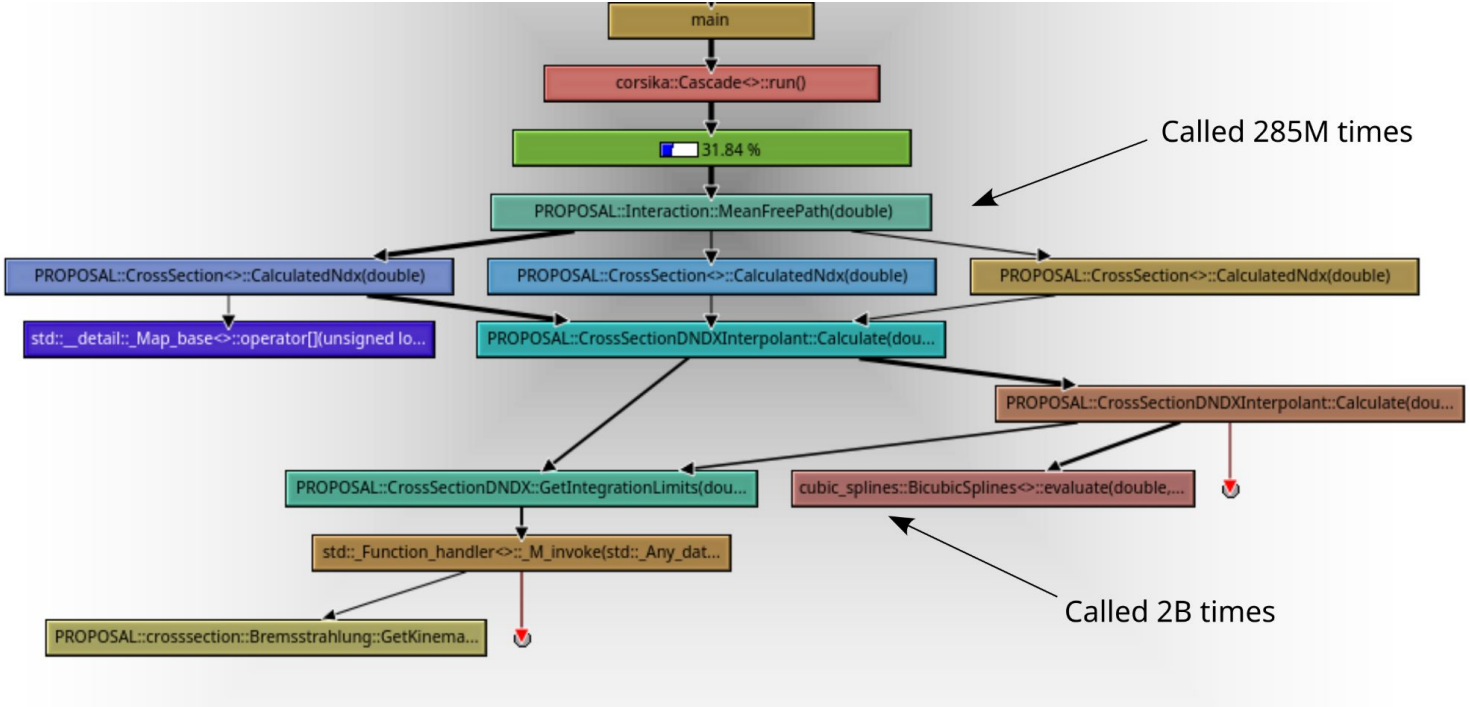





Callee Map for MeanFreePath, we can see that while MeanFreePath is being called 285M times, CubicSplines is being called 1.5B times, via MeanFreePath.

# Callcache information

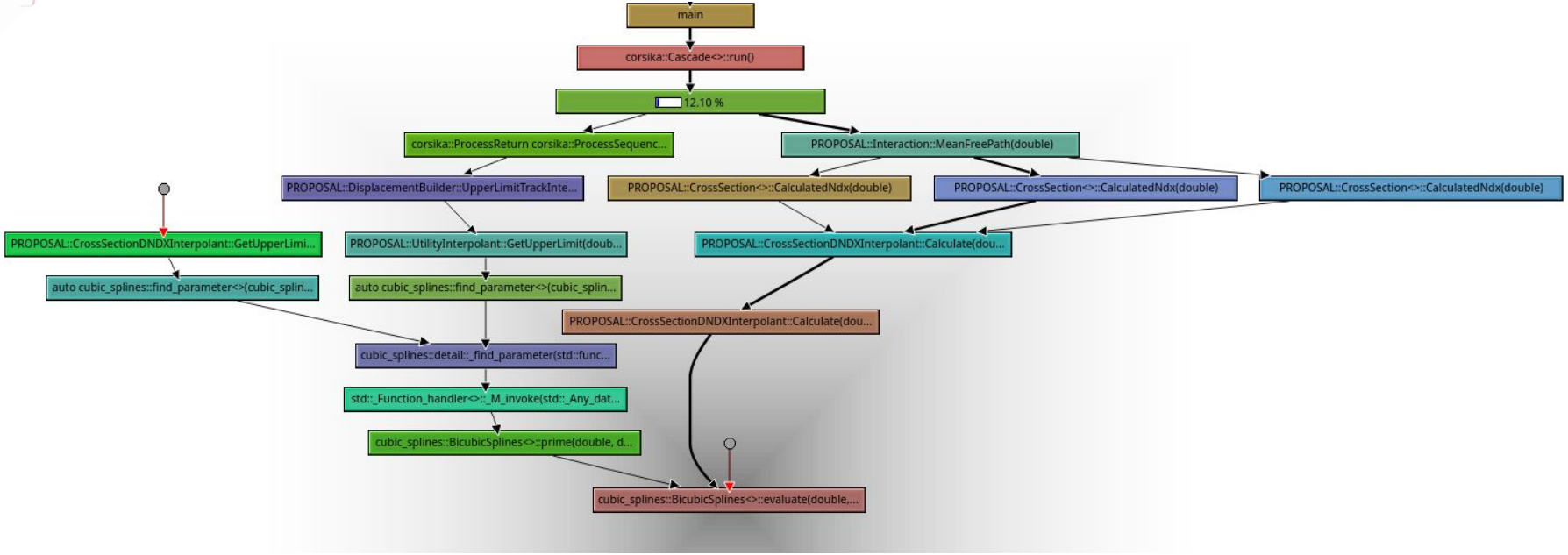





Callgraph illustrating the pathways between MeanFreePath and BiCubicSplies. The dominant pathway is highlighted.

# Callcache information

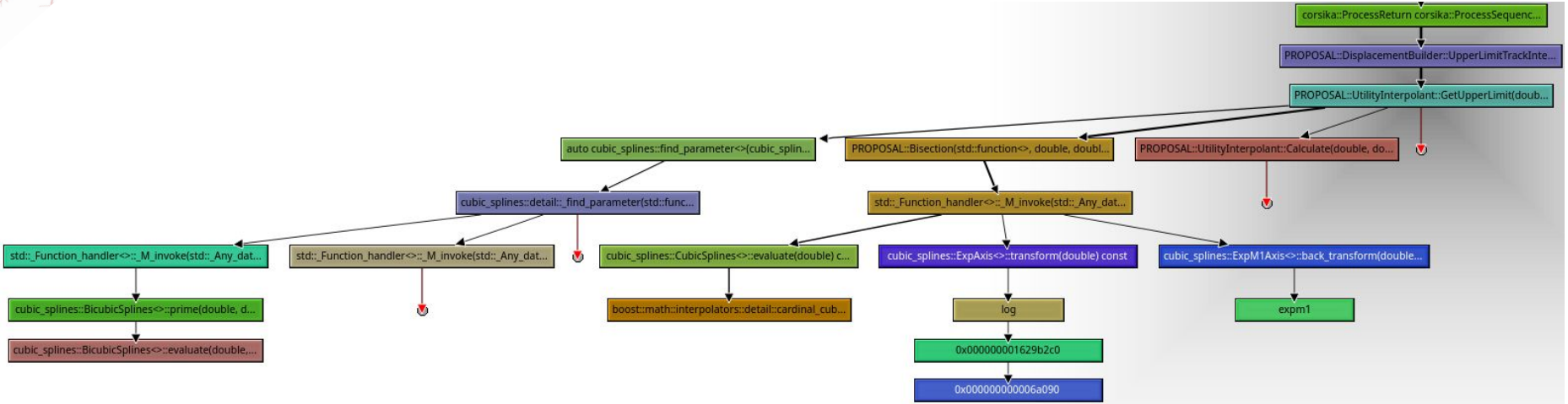





There are also other code pathways, which eventually lead to BiCubicSplies which contribute for the additional 0.5B times.

# Callcache information

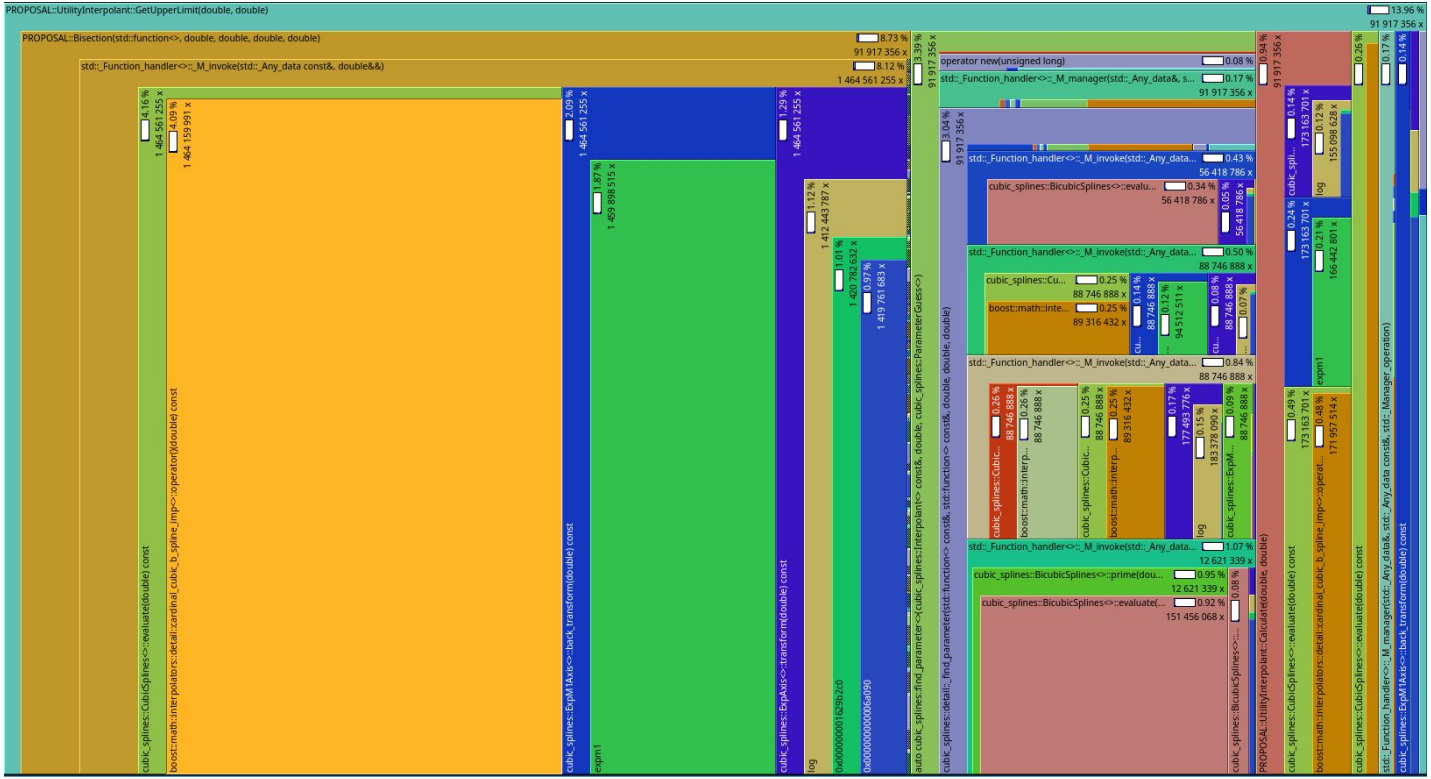





Callgraph for getUpperLimit, We dont see any immediate potential for improvement. The number of calls dont jump too much anywhere for us to spot infrastructural deficiencies.

# Callcache information

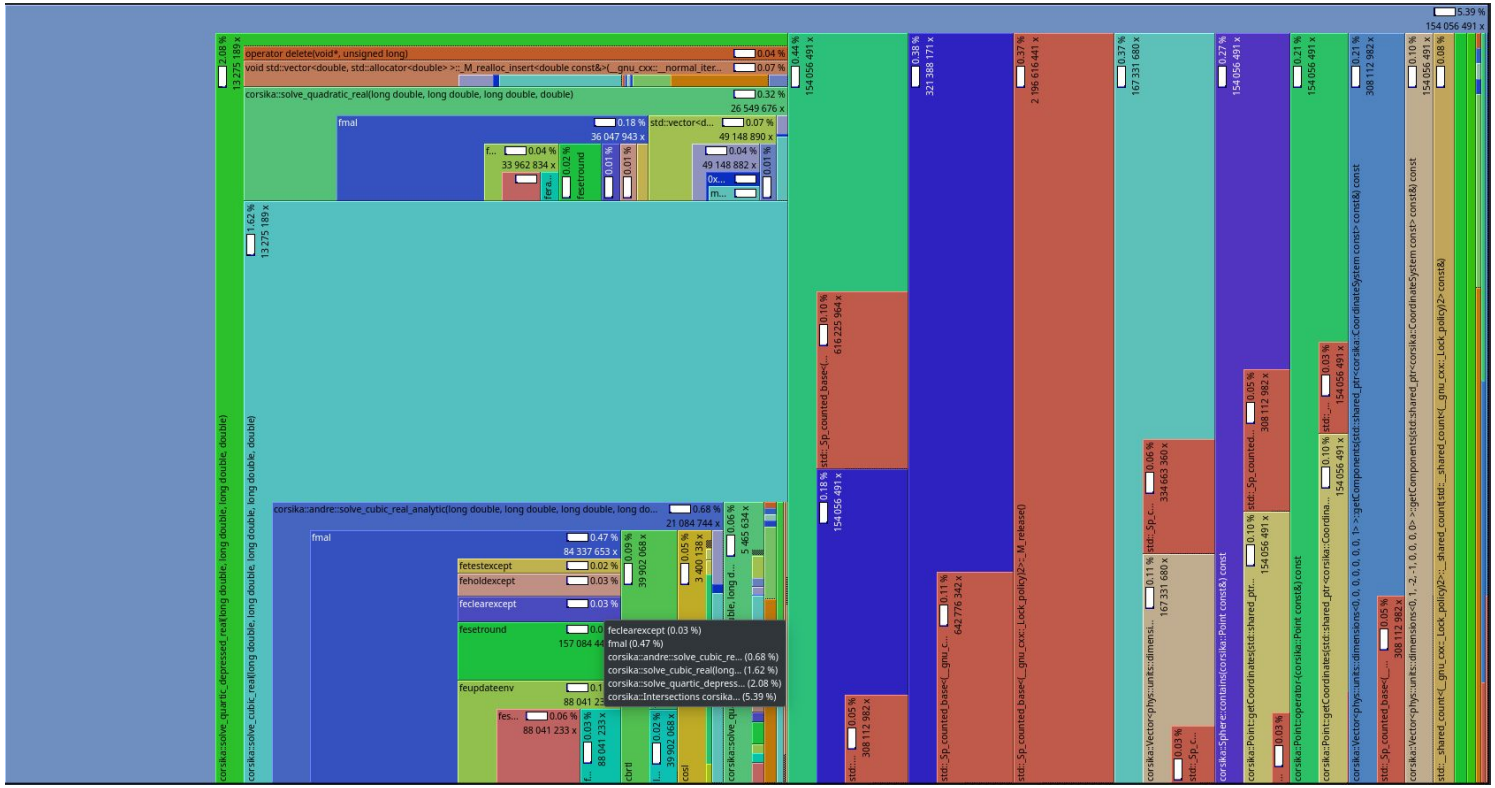





This probably calls for an algorithmic change, rather than an infrastructural change. Maybe some form of tabulation, which can reduce the calls to getUpperLimit.

# Callcache information

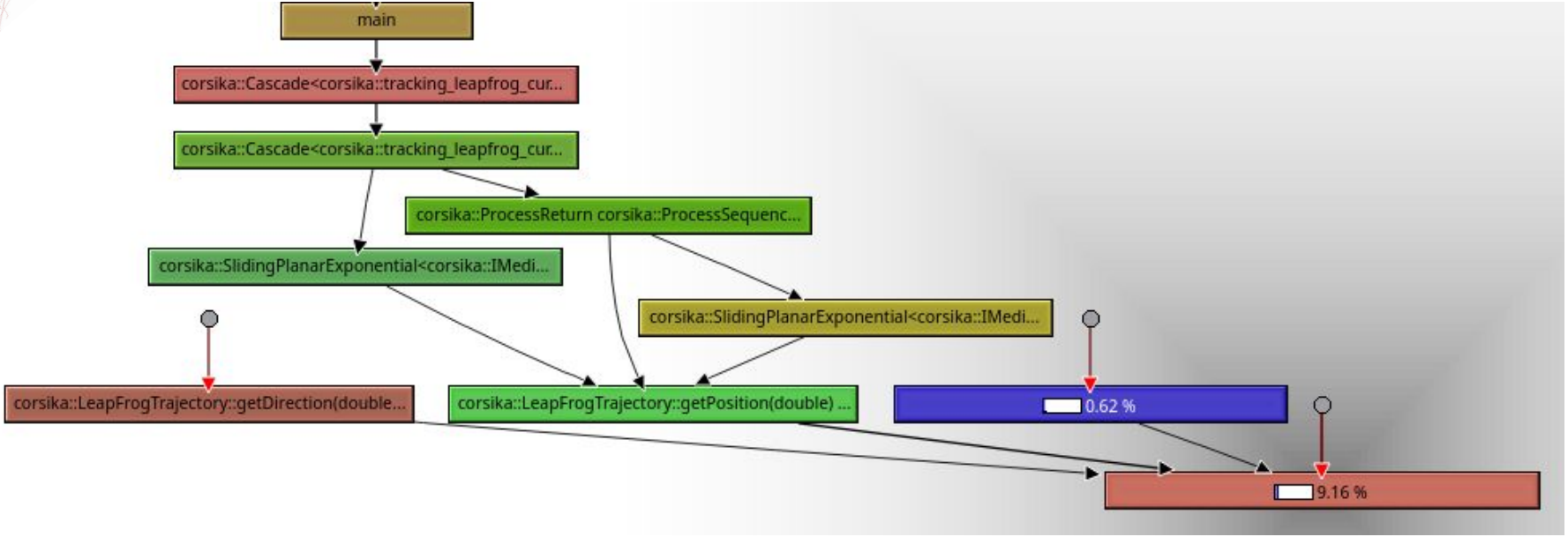





Another, hotspot in terms of timespent, is in the LeapFrog algorithm. There is probably potential for both infrastructural and algorithmic improvements here.

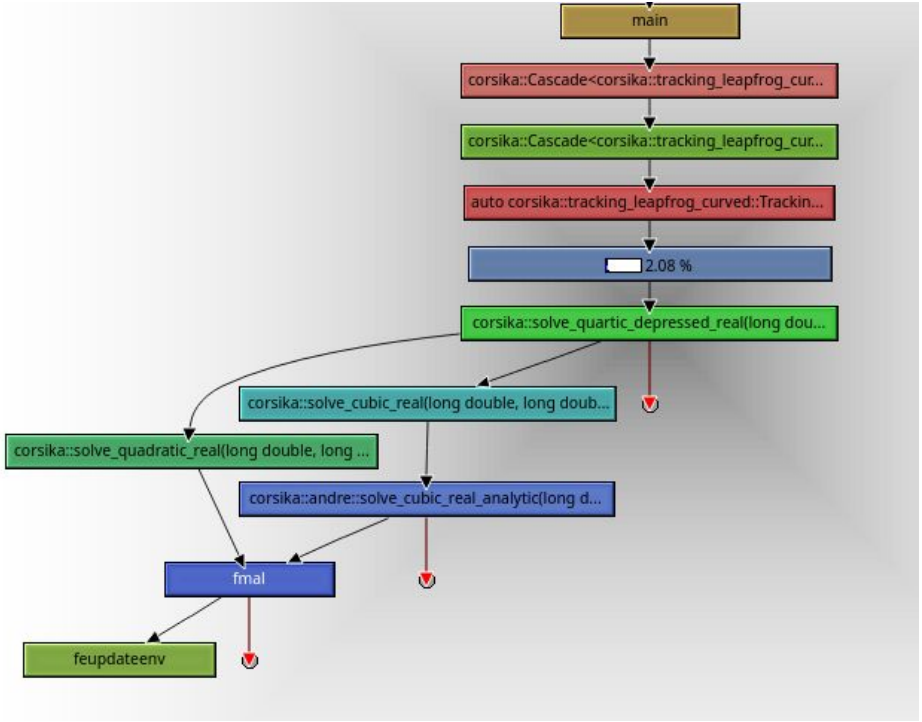
# Callcache information






At an infrastructure level, too many calls to memory management routines like M_release. Maybe better way to allocate and deallocate memory ? This was in release mode. So these are some non-trivial allocations which couldnt be optimized by the compiler

# Potential Improvement?




At an Algorithmic level, the cubic solver seems to take a decent chuck of time. We need to optimize it or avoid it altogether somehow.

# Summary

- Spotted three potential places for improvement, MeanFreePath, GetUpperLimit and LeapFrog
- MeanFreePath - Probably requires a infrastructural code change and a closer look into redundant calls.
- GetUpperLimit - Probably requires an algorithmic change, which helps us avoid these calls
- LeapFrog - Not really obvious what sort of change is required. It is probably some combination of both infrastructural and algorithmic change. So that we reduce memory allocations, and try to avoid the cubic solver or use it lesser.

Please, Playaround with these softwares yourself. A lot of insight can be gained by looking at the callgraph and the control-flow of the program.