Profiling Contur

Tools Used (1)

- We use cPython to profile the code which produces a .prof file. The profiling can be done with a small adjustment to the contur script in bin and then just running contur as normal from terminal.
- We use two separate packages to view the results in the .prof file, Snakeviz and gprof2dot.
- Snakeviz produces icicle and sunburst plots, it is browser based so has a dashboard style format where the user can dynamically play with the plots. See https://jiffyclub.github.io/snakeviz/ for further info.

Tools Used (2)

- gprof2dot produces static plots which show the flow of the code and the proportion of time spent in each part of the code, see https://github.com/jrfonseca/gprof2dot.
- All profiling results can be found in the following repository https://github.com/SeanBrayUCL/contur_profiles .
- Each profiling result in the repository consists of the .prof file which Snakeviz can be run on to see the profiling results and the png image which is the output of gprof2dot.

Changes Made So Far (1)

1. ErrMap Function (https://gitlab.com/hepcedar/contur/-/commit/5df35e7e2e3419fb650e3a2611338a9119865425) : We were calling the ErrMap function inside a for-loop, the update calls ErrMap just once outside the loop, saves the results in a dictionary and this dictionary is then called in the loop.
2. Test to statistic to confidence level (https://gitlab.com/hepcedar/contur/-/commit/0b80789580cd0529bfdce28d2bae252755c45a63): To convert the test statistic first to a p value we use scipy’s survival function. We were passing test statistics to this function individually, the change first collects test statistics into a numpy array and passes this numpy array into the scipy survival function.

Changes Made So Far (2)

3. Sort blocks, replacing list with dataframe (https://gitlab.com/hepcedar/contur/-/commit/407b0618b8538fe6491a5358b078900e7f01d1e9): For each pool we wanted the maximum confidence level of the analysis objects in the pool. We were doing this with a list and an if statement to restrict ourselves to analysis objects in a specified pool and then taking the max of the list. We were running this list concatenation inside another loop though which made things slow. Instead now we start with a dataframe of analysis objects pools and confidence level, group the dataframe by pool (using pandas groupby method) taking the max confidence level by pool.

Ongoing work

1. The change we made in update 2, passing numpy arrays instead of individual test statistics to the survival function can be taken further. Right now for each analysis object in a yoda file we will call this survival function three times, so for a whole yoda file we call it c.a. 3xnum_analysis_objects, two possible changes we can make here:
   - Reduce the number of calls in each analysis object from 3 to either 2 or 1, gains from doing this are likely very small, but may makes things cleaner;
   - Add an extra dimension to the numpy array storing test statistics for each analysis object in the numpy array and then passing this 3 dimensional numpy array to calculate p values for all yoda files in one call to the survival. This change likely offers a material performance improvement but requires more work to implement.

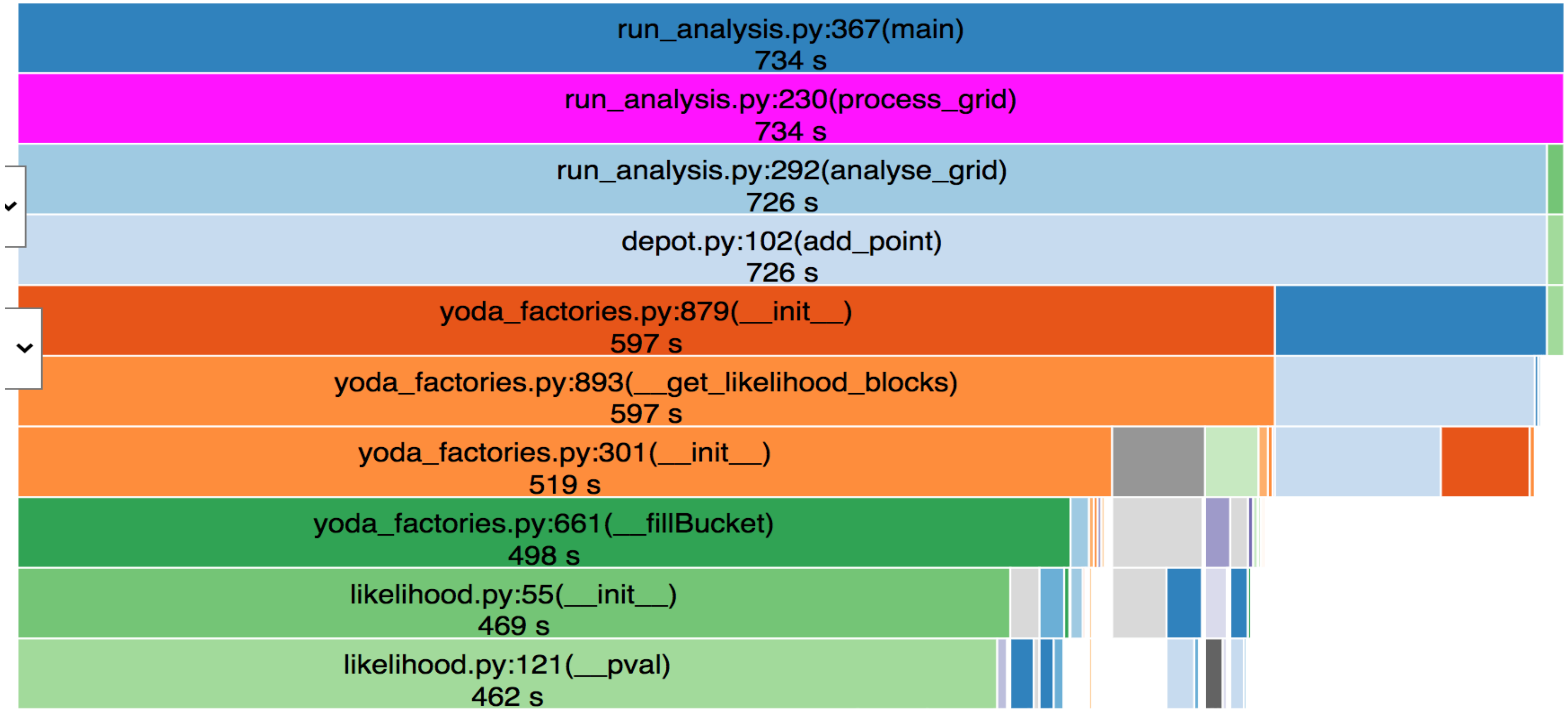
Ongoing work

2. Upgrade the suite of tests within Contur, this particularly important to ensure nothing is unintentionally broken when updating code to improve run time.

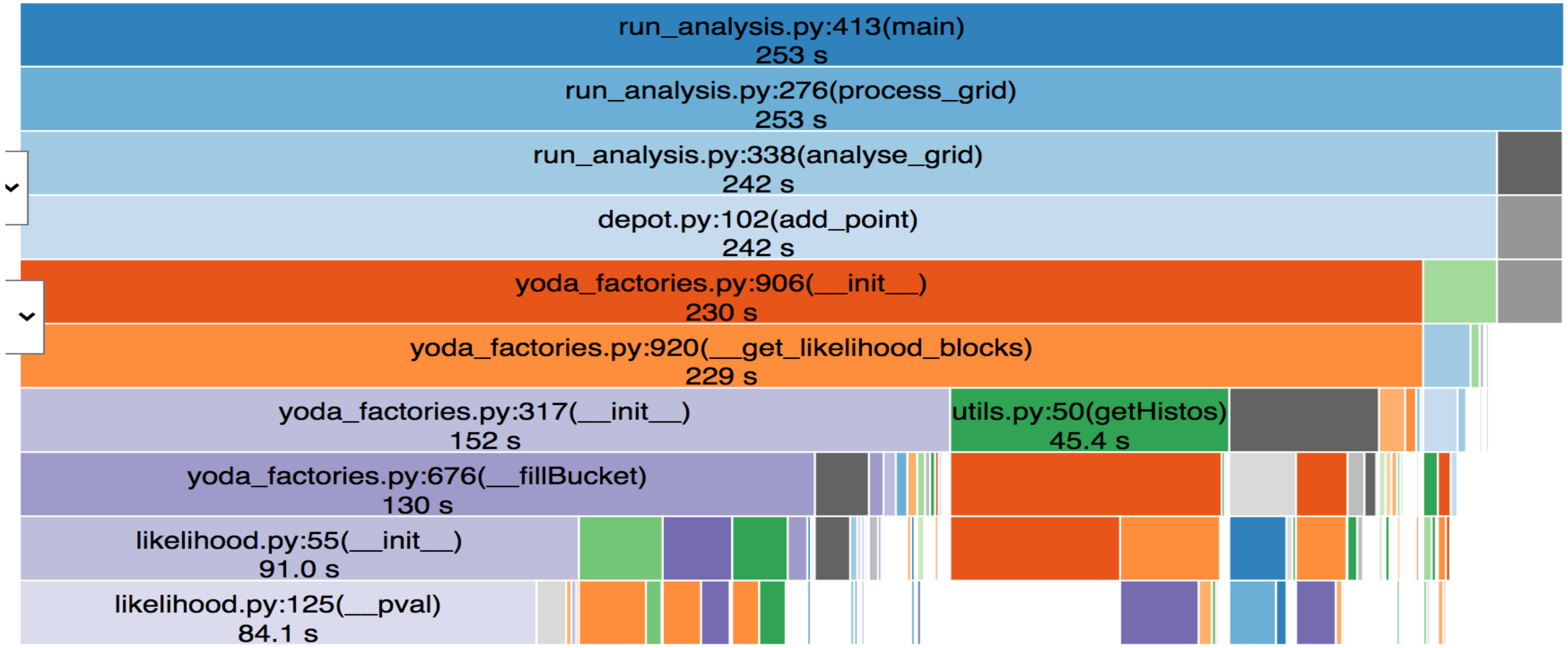
3. Look into parallelising the Contur grid run across yoda files. Currently the grid run effectively loops through each yoda file and combines results at the end into a single .map file.

Annex

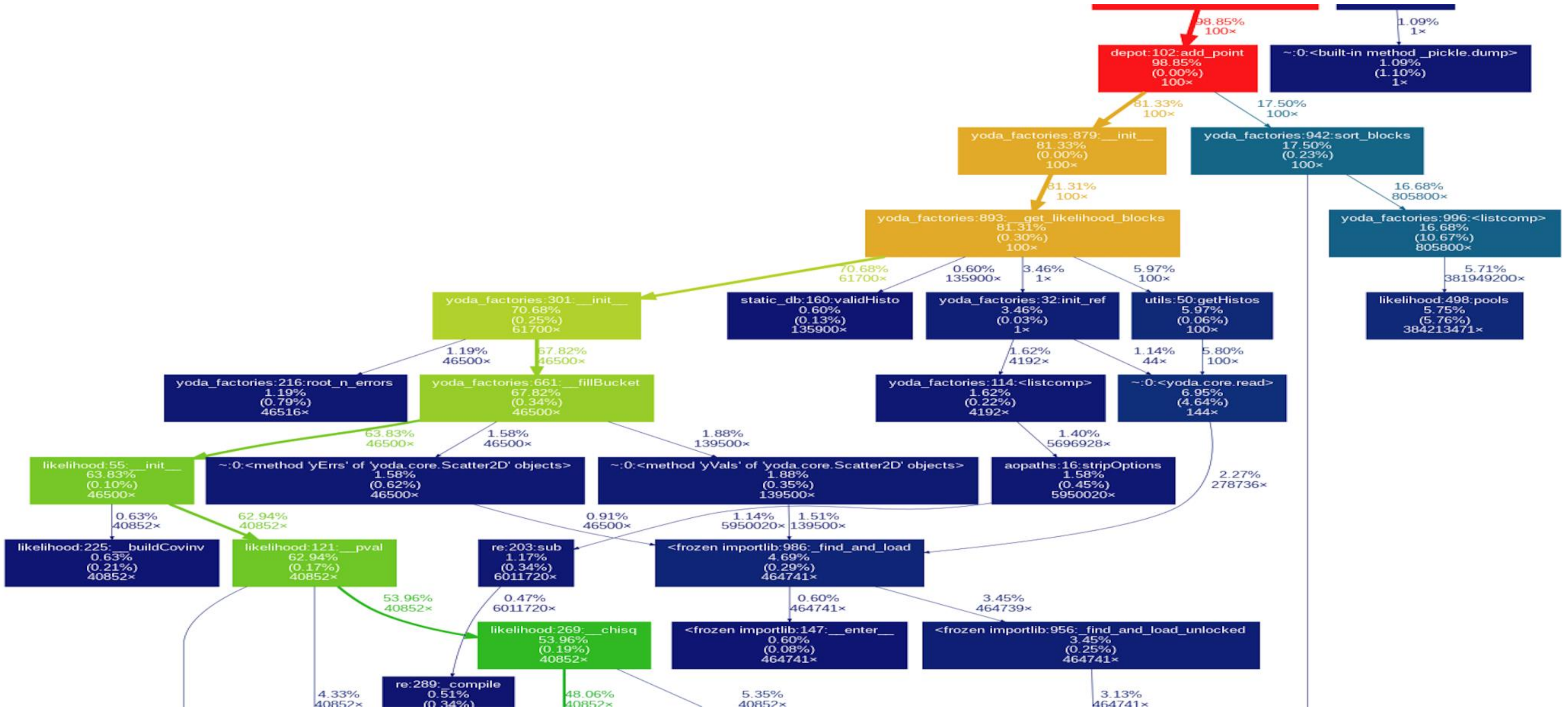
Grid Run (100 yoda files) Icicle Chart **before** change 2 and 3



Grid Run (100 yoda files) Icicle Chart **after** change 2 and 3

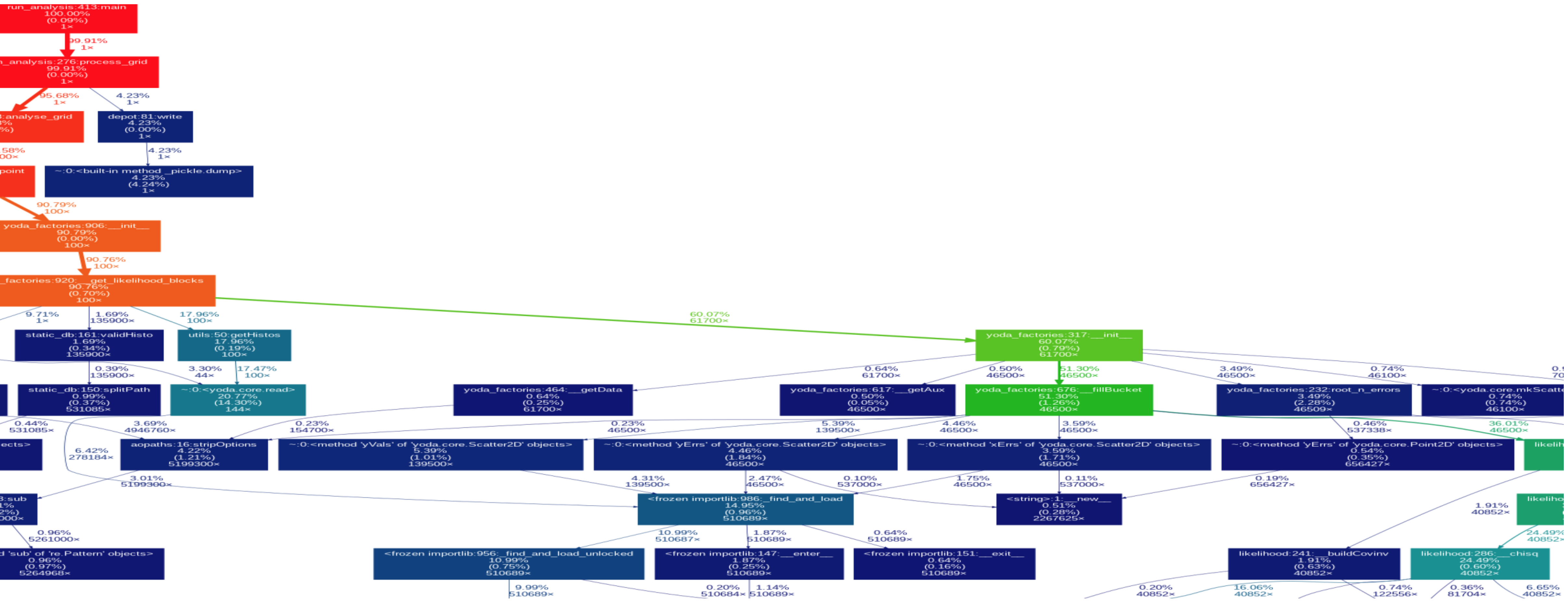


Grid Run (100 yoda files) gprof2dot Chart **before** change 2 and 3



depot:102:add_point 98.85% (0.00%) 100×
~:0:<built-in method _pickle.dump> 1.09% (1.10%) 1×
yoda_factories:879:__init__ 81.33% (0.00%) 100×
yoda_factories:942:sort_blocks 17.50% (0.23%) 100×
yoda_factories:893:__get_likelihood_blocks 81.31% (0.30%) 100×
yoda_factories:996:<listcomp> 16.68% (10.67%) 805800×
yoda_factories:301:__init__ 70.68% (0.25%) 61700×
static_db:160:validHisto 0.60% (0.13%) 135900×
yoda_factories:32:init_ref 3.46% (0.03%) 1×
utils:50:getHistos 5.97% (0.06%) 100×
likelihood:498:pools 5.75% (5.76%) 384213471×
yoda_factories:216:root_n_errors 1.19% (0.79%) 46516×
yoda_factories:661:__fillBucket 67.82% (0.34%) 46500×
yoda_factories:114:<listcomp> 1.62% (0.22%) 4192×
~:0:<yoda.core.read> 6.95% (4.64%) 144×
likelihood:55:__init__ 63.83% (0.10%) 46500×
~:0:<method 'yErrs' of 'yoda.core.Scatter2D' objects> 1.58% (0.62%) 46500×
~:0:<method 'yVals' of 'yoda.core.Scatter2D' objects> 1.88% (0.35%) 139500×
aopaths:16:stripOptions 1.58% (0.45%) 5950020×
likelihood:225:__buildCovinv 0.63% (0.21%) 40852×
likelihood:121:__pval 62.94% (0.17%) 40852×
re:203:sub 1.17% (0.34%) 6011720×
<frozen importlib:986:_find_and_load 4.69% (0.29%) 464741×
likelihood:269:__chisq 53.96% (0.19%) 40852×
<frozen importlib:147:__enter__ 0.60% (0.08%) 464741×
<frozen importlib:956:_find_and_load_unlocked 3.45% (0.25%) 464741×
re:289:_compile 0.51% (0.34%)
98.85% 100×
1.09% 1×
81.33% 100×
17.50% 100×
81.31% 100×
16.68% 805800×
70.68% 61700×
0.60% 135900×
3.46% 1×
5.97% 100×
5.71% 381949200×
1.19% 46500×
67.82% 46500×
1.62% 4192×
1.14% 44×
5.80% 100×
63.83% 46500×
1.58% 46500×
1.88% 139500×
1.40% 5696928×
2.27% 278736×
0.63% 40852×
62.94% 40852×
0.91% 46500×
1.14% 5950020×
1.51% 139500×
53.96% 40852×
0.47% 6011720×
0.60% 464741×
3.45% 464739×
4.33% 40852×
48.06% 40852×
5.35% 40852×
3.13% 464741×

Grid Run (100 yoda files) gprof2dot Chart **after** change 2 and 3


run_analysis:413:main 100.00% (0.09%) 1×
99.91% 1×
_analysis:276:process_grid 99.91% (0.00%) 1×
95.68% 1×
4.23% 1×
:analyse_grid
depot:81:write 4.23% (0.00%) 1×
4.23% 1×
~:0:<built-in method _pickle.dump> 4.23% (4.24%) 1×
90.79% 100×
yoda_factories:906:__init__ 90.79% (0.00%) 100×
90.76% 100×
_factories:920:__get_likelihood_blocks 90.76% (0.70%) 100×
60.07% 61700×
yoda_factories:317:__init__ 60.07% (0.79%) 61700×
static_db:161:validHisto 1.69% (0.34%) 135900×
utils:50:getHistos 17.96% (0.19%) 100×
static_db:150:splitPath 0.99% (0.37%) 531085×
~:0:<yoda.core.read> 20.77% (14.30%) 144×
yoda_factories:464:__getData 0.64% (0.25%) 61700×
yoda_factories:617:__getAux 0.50% (0.05%) 46500×
yoda_factories:676:__fillBucket 51.30% (1.26%) 46500×
yoda_factories:232:root_n_errors 3.49% (2.28%) 46509×
~:0:<yoda.core.mkScatte 0.74% (0.74%) 46100×
aopaths:16:stripOptions 4.22% (1.21%) 5199300×
~:0:<method 'yVals' of 'yoda.core.Scatter2D' objects> 5.39% (1.01%) 139500×
~:0:<method 'yErrs' of 'yoda.core.Scatter2D' objects> 4.46% (1.84%) 46500×
~:0:<method 'xErrs' of 'yoda.core.Scatter2D' objects> 3.59% (1.71%) 46500×
~:0:<method 'yErrs' of 'yoda.core.Point2D' objects> 0.54% (0.35%) 656427×
<frozen importlib:986:_find_and_load 14.95% (0.96%) 510689×
<string>:1:__new__ 0.51% (0.28%) 2267625×
<frozen importlib:956:_find_and_load_unlocked 10.99% (0.75%) 510689×
<frozen importlib:147:__enter__ 1.87% (0.25%) 510689×
<frozen importlib:151:__exit__ 0.64% (0.16%) 510689×
d 'sub' of 're.Pattern' objects> 0.96% (0.97%) 5264968×
likelihood:241:__buildCovinv 1.91% (0.63%) 40852×
likelihood:286:__chisq 24.49% (0.60%) 40852×