gamma-speed Documentation

Release 0.1

Andrei Ignat & Christoph Deil

August 08, 2013

CONTENTS

The **gamma-speed** repository can be found here.

gamma-speed is a project meant to profile and monitor ctools and gammalib.

The motivation behind this project was that as new tools are being developed, the developer must see if the tools are working properly. A few different possibilities of monitoring these tools were considered. After a bit of online research, the conclusion was that given the limited time frame(8 weeks) the best solution would be to build Python based tools.

In order to monitor different system resources, the Python module `psutil` is being employed. For more details about how it works, see the psutil website.

Different `psutil` functionalities are put together in monitor in order to get informaton about the system running a certain process. Monitor gathers this information and writes it to disk. After this, another python module will interpret the information coming from the output of monitor and plot the results.

A key point in this project was seeing how well `gammalib` and `ctools` parallelism obeys Amdahl's Law. This law puts a limit on the maximum speed-up that can be obtained through parallelism and will be described in a later chapter.

In the end, a few conclusions were drawn about the parallel behaviour of these tools and an outline for future improvements to the project was drawn.

Contents

CHAPTER

ONE

PROJECT DESCRIPTION

- Student: Andrei Cristian Ignat
- Supervisor: Dr. Christoph Deil
- Time: June 10, 2013 to August 9, 2013 (9 weeks)
- Place: MPIK Heidelberg, H.E.S.S. group of Prof. Werner Hofmann

1.1 Abstract

Astronomical gamma-ray data analysis can be very CPU and / or I/O intensive. The purpose of this 9-week, first year physics student project is to time and profile typical data analysis tasks with a focus on the speedups that can be obtained for the maximum likelihood fitting step by using multiple CPU cores.

1.2 Introduction / Overview / Links

This is basically an introduction and link collection for Andrei. The project description is in the next section.

- To get an overview of gamma-ray astronomy science, read one of the review articles listed here.
- We will analyse data from two types of instruments:
- Fermi is a satellite observing in the GeV energy range. It's data and software (the Fermi Science tools) are publicly available here.
- Imaging atmospheric Cherenkov telescopes (IACT) arrays are ground-based and observe in the TeV energy range. Currently there are three telescope arrays in operation (HESS, MAGIC, VERITAS and one in preparation: CTA. No data is publicly available, so we'll probably use simulated HESS or CTA data sets.
- Fermi GeV data analysis and IACT TeV data analysis are pretty different.
- Fermi has a very large field of view (about 10% of the whole sky at any time) and is continuously observing the sky. It has detected about 2000 point-like sources (see the 2FGL catalog) and it's main background for source detection is Galactic diffuse gamma-ray emission (see this paper).
- IACTs have a small field of view (a few degrees across) and the data consists of pointed observations (called "runs") on sky positions near suspected source positions. About 150 TeV sources have been detected so far (see TeVCat) and the main background is from charged cosmic ray particles (see Berge (2007)). A good introduction to TeV astronomy and analysis methods is de Naurois (2012).
- Nevertheless the last analysis step of modeling / fitting the data is very similar for Fermi and IACTs.

- The data either consists of unbinned event lists (energy, RA, DEC) or binned count cubes (events filled into energy-RA-DEC bins) from some small region of the sky, the so-called "region of interest (ROI)", which for Fermi typically is a circle with radius 10 to 20 deg.
- The other input into the fit is observation exposure (computed from the instrument effective area and exposure time) and instrument resolution information. The spatial resolution is given by the so-called point spread function (PSF), the energy resolution is given by a matrix, see e.g. the Fermi LAT instrument response functions (IRFs) here.
- The model consists of a background model plus a number of sources, each with a spatial and spectral model with free parameters (see Sections 7.5.1 and 7.6.1 in de Naurois (2012)).
- The model is then fit to the data taking the exposure and instrument resolution into account in a likelihood fit (see description for Fermi here or here and Chapter 7 in de Naurois (2012) for IACTs). Unfortunately I'm not aware of a short and simple intro to likelihood fitting for beginners, maybe you can read this or this or do some of the tutorial from Sherpa or iminuit or probfit or RooFit or the python4astronomer's Sherpa fitting tutorial. Understanding the steps involved in computing the likelihood function (a.k.a. "cost function" for "fit statistic") for a given set of model parameters will be the hardest part of this project, but it is probably necessary to a certain degree to understand where (numerical integration or convolution or model evaluation or computation of derivatives or summation over events / bins or ...) the CPU time is spent when profiling the analysis.
- We will be using the following software:
- The main focus will be on gammalib and ctools and specifically the ctlike tool. The ctools can analyze Fermi data and, to a certain degree, HESS and CTA data analysis. They use OpenMP to run faster on multi-core machines.
- We might compare the ctlike speed against gtlike in the Fermi science tools (can only use one core as far as I know).
- If we also look at other steps like exposure computation or diffuse response or TS map computation, we might also look at gtapps_mp, which uses Python multiprocessing to use multiple CPU cores.
- If there is time, we might look at the speed of the HESS analysis program [HAP], which is a HESS internal analysis software.
- If there is time, we might look at the speed of morphology fits with Sherpa, which also uses Python multiprocessing to achieve speed-ups for certain steps (parameter error computation) as described here.

1.3 Project plan

We have nine weeks ... it's very hard to predict how fast results are obtained ... so I reserved week 6 to continue with the main project or to do one of the side projects and there are two weeks at the end to write up the report and finish up loose ends.

- Week 1: Learn some gamma-ray astrophysics (see references above)
- Week 2: Learn some gamma-ray data analysis methods (see references above)
- Week 3: Define and produce test data sets (one Galactic and one extra-galactic; one Fermi and one HESS)
- Week 4: Run and time analyses with ctools on at least two machines and measure the speedup with the number of cores.
- Week 5: Profile the analyses to find out where the CPU time is spent. Possibly try different compilers (gcc, clang, icc) and optimiser flags.
- Week 6: Continue main project or if there is time do one of these things: time HAP, gt_apps_mp or Sherpa (see above) or some of the other ctools tasks (see here). Looking at CPU usage, memory usage and disk I/O would

also be interesting to get a rough overview of what the analyses are doing (e.g. ctselect speed is probably disk I/O speed limited)

- Week 7: Write up report
- Week 8: Iterate project report (e.g. clearer description or double-check results or add additional plots or ...)

The project report and notes and scripts in the https://github.com/gammapy/gamma-speed/ repo are the product of your project. It should be a starting point for further work on HESS, Fermi, CTA data analysis speed by others in the future. Detailed descriptions of which tools you tried to time and profile (and possibly measure memory usage and disk I/O) and which are useful and which aren't and how to use them is helpful.

The most useful thing would be an automatic script that measures certain aspects of ctools performance for typical analysis scenarios that can easily be re-run to try out speed improvements and prevent performance regressions, but this level of automation is most likely not possible in the given time. Just to get the idea I have in mind here, have a look at the PyPy speed center or the pandas benchmark as measured by vbench

1.4 Further references

Here's some more useful references for tools you might use:

- Learn git: basic tutorial, advanced tutorial
- Learn Python astro basics: https://astropy4mpik.readthedocs.org/ and http://python4astronomers.github.io
- Profiling tool: KCachegind

CHAPTER

TWO

AMDAHL'S LAW

2.1 Theory

...states that, given *P* cores, the speed up for that number of cores is *S(P)*

$$S(P) = \frac{t_{total}(1)}{t_{total}(P)} = \frac{t_{serial} + t_{parallel}}{t_{serial} + \frac{t_{parallel}}{P}}$$

with t_{total} the total runtime, t_{serial} the time needed to run the serial part of the code and $t_{parallel}$ the time needed to run the parallel part of the code.

Under this formulation, the maximum speedup, as $P \to \infty$ is

$$S_{max} = \frac{t_{total}}{t_{serial}} = 1 + \frac{t_{parallel}}{t_{serial}}$$

2.2 Applying the law

- the output of `monitor` for 1 CPU contains the total runtime i.e. $t_{serial} + t_{parallel}(1)$
- log statements have been added to the source code such that the parallel runtime $t_{parallel}(P)$ is recorded for *P* CPU's

Thus, you obtain the predicted speed up $S_p(P)$ and the observed speedup $S_o(P)$

$$S_p(P) = \frac{t_{serial} + t_{parallel}(1)}{t_{serial} + \frac{t_{parallel}(1)}{P}} \qquad S_o(P) = \frac{t_{serial} + t_{parallel}(1)}{t_{serial} + t_{parallel}(P)}$$

From these, one can derive the efficiency for *P* cores as

$$E_p(P) = \frac{S_p(P)}{P} \qquad E_o(P) = \frac{S_o(P)}{P}$$

External resources

Wikipedia

Reevaluating Amdahl's Law and Gustafson's Law

CHAPTER

THREE

CPU INFOS

This is a short summary description of the machines used for this study.

3.1 Andrei's Asus G51J

- Processor: Intel Core i5 430M (Intel Info)
- Cache: L2: 3072 kB
- Memory: 2 x 2 GB DDR3 @ 1067MHz
- Graphics: NVIDIA GeForce GTS 360M 1024MB (can be used with CUDA)
- Storage: Momentus ST9500420AS
- OS: Ubuntu 12.10(quantal) 64bit

3.2 Andrei's Desktop machine

- Processor: 3.0 GHz Intel Core 2 Duo E8400(2 cores)
- Cache: L2: 6 MB
- Memory: 1 GB DDR3 @ 1066 MHz + 2 GB DDR3 @ 1066 MHz
- Graphics: Intel Graphics Media Accelerator 950
- Storage: Seagate ST1000DM003-9YN162 1TB
- OS: Ubuntu 12.04(precise) 64bit

3.3 New MPIK cluster machine

- Name: `lfc301`
- Processor: 2 x Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz = 16 cores (Intel Info)
- check: 2.2 GHz E5 2660 -> Turboboost 2.8 GHz if only some cores are used
- Cache: 20 MB
- Memory: 132 GB
- Storage: Lustre file system

- OS: Scientific Linux 6

3.4 Old MPIK cluster machine

- Name: `lfc292`
- Processor: 2 CPU x Intel(R) Xeon(R) CPU E5450 @ 3.00GHz = 8 cores (Intel Info)
- check: 3 GHz
- Cache: 6 MB
- Memory: 16 GB
- Storage: Lustre file system
- OS: Scientific Linux 6

3.5 Christoph's Macbook Pro Retina

- Processor: 2.6 GHz Intel Core i7-3720QM (4 core) (Intel Info)
- Cache: L2: 256 KB per core, L3: 6 MB
- Memory: 2 x 8 GB = 16 GB 1600 MHz DDR3
- Graphics: NVIDIA GeForce GT 650M 1024 MB (can be used with CUDA)
- Storage: APPLE SSD SM512E
- OS: OS X 10.8.4

3.5.1 Howto

Here are some notes on how to find out more about a given machine:

- `cat /proc/cpuinfo`
- `df -h`
- `sudo dmidecode --type 17` for RAM information
- Get CPU Info on a Mac: `sysctl -n machdep.cpu.brand_string`

Note: The results presented later were obtained from the new MPIK cluster machine `lfc301`.

CHAPTER

FOUR

HOW TO INSTALL THE NECESSARY SOFTWARE

In order to be able to run `monitor.py` on your system, you are going to need to install a few dependencies.

4.1 Python dependecies

First of all, you need at least **python 2.7** or later in order to be able to `import argparse`.

After you have solved this issue, `monitor.py` is heavily based on the following packages:

- psutil - for obtaining information about the resources being used by the system;
- pandas - for saving the information obtained from psutil and the plotting it later on;
- matplotlib - for plotting the different results.

Usually this is only a matter of simply running

```
pip install <PACKAGE_NAME>
```

Besides these basic requirement, in order to monitor `ctools` and `gammalib` you need to install them first :D

4.2 `ctools` and `gammalib`

In order to profile and benchmark `ctools` and `gammalib`, one obviously needs to have installed these two pieces of software. In the monitoring process though, the need arose to insert logging statements in the source code. Therefore, a slightly modified version was used. This version can be found at

- gammalib
- ctools

In order to make things easier though, there is a script that will automatically install these tools for you. This is a shell script that runs on Unix machines and assumes that you have already installed github on your machine. Running it is an extremely simple matter of going to the `gamma-speed/scripts` directory, copying `install_ctools.sh` to the directory in which you want to install `ctools` and `gammalib` and running it under the command:

```
./install_ctools.sh LOG gammaspeed_extra_log
```

Note: If you want to install the normal version of `gammalib` and `ctools`, you can run `./install_ctools.sh NORMAL`. However, all the examples that will be presented from now on were run with the extra logging version of

the gammalib and ctools. If you still wish to install the normal version, you should be aware that in the following examples, whenever you see the option -log=True, you should replace it with -log=False when executing python scripts from gamma-speed.

4.3 Logging in gammalib

If you have installed the software using:

```
./install_ctools.sh LOG gammaspeed_extra_log
```

then you have the version that implements the following way of logging in gammalib. If you want to add secondary log statements, just follow the method described below.

An extremely useful tools for logging comes from gammalib and is called GLog. This class provides the user a simple and easy to use log object. After importing the header for the logger class

```
#include "GLog.hpp"
```

one cam simply create a new logger object and start logging away!:

```
GLog logtt;
clobber = true;
filename = "magicLog.log"
logtt.open(filename, clobber);
logtt.date(true);
logtt << "logger: at your service" <<std::endl;
logtt.close();
```

Of course, there are fancier options available from the GLog class such as different headers, formats, etc. For debugging purposes though, the little piece of code above will suffice.

4.4 Setting up your environment

After you have installed all these magnificent pieces of software, you still need to tell your machine that they are there. My reccomendation would be to add the following lines of code at the end of your .bashrc file. Obviously, you can export these variables everytime you want to monitor something or can build a separate script that contains these lines of code and run this script each time before starting to monitor.

1. In order to tell your machine that you have installed the monitor, you need to add the scripts/src directory to your PYTHONPATH. This is done by calling:

   ```
   export PYTHONPATH=$PYTHONPATH:<path/to/gamma-speed>/gamma-speed/scripts/src
   ```

   Now, your system knows that there are Python modules that can be imported in the src directory.

2. If you want to monitor gammalib and ctools, you need to tell your system that you have installed these two and where to look for them. This is done by running the following lines of code:

   ```
   export GAMMALIB = <INSTALL_DIR>
   export CTOOLS = <INSTALL_DIR>
   source $CTOOLS/bin/ctools-init.sh
   source $GAMMALIB/bin/gammalib-init.sh
   ```

   Now you can run the ctools tests that are in the gamma-speed repository.

CHAPTER

FIVE

TUTORIAL

In this short tutorial, a typical example of how to run the `monitor` is shown. After the measurement is over, a typical usage of the `MonitorPlot` class is described.

Assuming that you have added the `scripts/src` folder to your `PYTHONPATH`, you can now import the `monitor` and `monitor_plot` modules inside Python scripts. This is what we will also do.

Note: Except for the pieces of code where **Amdahl's Law** is required, none of the following examples require that the source code be modified to include logging statements that tell monitor how much time the process is spending in executing serial and parallel code.

5.1 Monitoring

Start a new Python script called `monitor_process_name` and add:

```
import monitor as mt
```

to the import list of the script.

Now, in order to be able to run and monitor the process, you will need to instantiate a new `Monitor` object. This is done by simply calling:

```
process_monitor = mt.Monitor("<process>", number_of_threads)
```

This object will now start the command you have passed to it and run it on `number_of_threads` threads. In order to actually gather information about the resources that the process is using, you need to call the `monitor` like this:

```
process_monitor.monitor("monitor_CPUs_" + str(number_of_threads) + ".csv", 0.1)
```

The arguments passed to the `monitor` method dictate the name of the `outfile` and the `cpuinterval` or frequency with which the measurements should take place.

Note: If you have added logging statements to the `ctools` source code, you may also call the following method:

```
process_monitor.parse_extension(logext='*.log', outname='process_CPUs_' + str(nthrd + 1) + '.csv', ti
```

in order to select your logging statements and save them to separate outfiles.

5.2 Plotting

Plotting your results is a task for the `MonitorPlot` class. First, import the Python module containing it with:

```
import monitor_plot as mtp
```

Now, you will need to instantiate a `MonitorPlot` object from this class. This is easily done by calling:

```
my_plotter = mtp.MonitorPlot(args.infile, args.nrcsv, " function name")
```

This will tell the plotter what type of pattern the `.csv` filenames follow and also how many `.csv` files it should read.

The `my_plotter` object can now call methods that produce different plots. You may want to produce separate plots for CPU, RAM, disk READ and disk WRITE or you might to plot all these values on a single figure. These can be accomplished by calling:

```
#For separate plots
my_plotter.CPU_plot('CPU.png')
my_plotter.MEM_plot('MEM.png')
my_plotter.IO_speed_plot('io_write.png')
my_plotter.IO_read('io_read.png')

#For a all the plots in a single figure
my_plotter.mplot(outfile='all_plots.png', figtitle='Resource utilization for process')
```

The second thing that the `my_plotter` object can do is plot the measured speedup from executing the process on multiple cores. In order to do this, it needs to know the time needed to execute the process on a single core and then the execution times on multicore machines. There are three different kinds of plots that can be obtained, namely

- a bar plot with the execution time versus number of cores - using the function `times_bar`
- the speedup versus number of cores - using the function `speed_plot`
- the efficiency of the process versus number of cores - using the function `eff_plot`

The values that are used for these plots are obtained from the start and end times read from the normal output of monitoring. Again, you can choose to have all these plots separately or together in one figure. In order to produce the plots, call:

```
# first read the values from the monitor output
times = pd.Series(index=cores)

for i in xrange(my_plotter.ncsv):
        df = my_plotter.read_monitor_log(i)
        # time spent for the whole process
        times[i + 1] = df['TIME'].iget(-1) - df['TIME'].iget(0)

# for separate plots
my_plotter.times_bar('time_bar_plot.png', ax=None, speed_frame=times)
my_plotter.speed_plot('speed_up_plot.png', ax=None, speed_frame=times, amdahl_frame=None)
my_plotter.eff_plot('eff_up_plot.png', ax=None, speed_frame=times, amdahl_frame=None)

# for all the plots in one figure
my_plotter.splot(figtitle='Speedup analysis for process', outfile='splot.png', speed_frame=times, amd
```

5.2.1 Amdahl's Law

A functionality of plotting is that you can choose to add the speedup values predicted by Amdahl's Law to the plots. For this, you have to know how much time is spend executing the parallel code of the process `parallel_time[P]`,

where P is the number of cores used. In code, this would look like:

```
amd = pd.Series(index=cores)
parallel_loop = pd.Series(index=cores)

for i in xrange(my_plotter.ncsv):
    df = my_plotter.read_monitor_log(i)
    start_time = df.at[0, 'TIME']
    aux = select_lines('secondary_log_CPUs_' + str(i + 1) + '.csv',
        start_time,
        ['gammaspeed:parallel_start',
         'gammaspeed:parallel_end'])
    parallel_time[i + 1] = aux[1] - aux[0]

parallel_time_init = parallel_loop[1]

for i in xrange(my_plotter.ncsv):
    amd[i + 1] = times[1] / (times[1] - parallel_time_init + parallel_time_init / (i + 1))
```

where `'secondary_log_CPUs_' + str(i + 1) + '.csv'` is the `.csv` file that contains the two timestamps for `gammaspeed:parallel_start` and `gammaspeed:parallel_start`, i.e. how much time was spent in the execution of the parallel portion of the code. Addind these values to the previous plot is done by simply adding the `amd` Series as an argument when calling the `splot` function:

```
# for separate plots
my_plotter.speed_plot('speed_up_plot.png', ax=None, speed_frame=times, amdahl_frame=amd)
my_plotter.eff_plot('eff_up_plot.png', ax=None, speed_frame=times, amdahl_frame=amd)

# for all the plots in one figure
my_plotter.splot(figtitle='Speedup analysis for process', outfile='splot.png', speed_frame=times, amd
```

CHAPTER

SIX

RESULTS

The results are structured in what was obtained for the measurements of `ctobssim`, `ctlike` and the `pipeline` analysis. There is a brief introduction before each of these results are presented.

The parallel parts of the monitored functions have been parallelized using OpenMP. Monitor was used to set the environment variable `OMP_NUM_THREADS` to different values in order to insure that the software could not access more than a specified number of cores.

In order to get the time spent in the serial and parallel parts of the programme, logging statements were added to the source code, statements which logged the duration of the parallel portion. These values were then interpreted according to Amdahl's Law and the theoretical speedup was ploted together with the measured values.

6.1 `monitor` checks

Before the actual monitoring begins, a few simple test cases have been used to ee if the monitor actually does its job.

6.1.1 CPU usage test

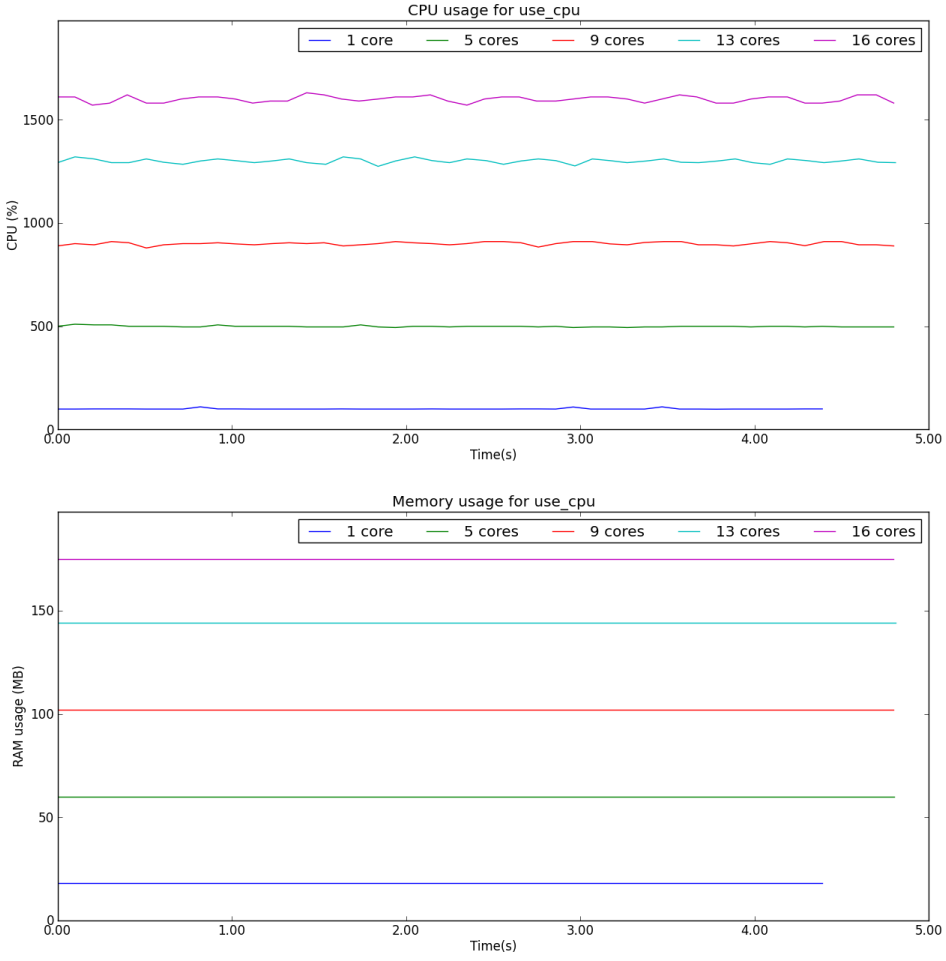
The CPU usage test serves the purpose of testing if `monitor` can actually set the number of threads that the program is using i.e. if it correctly exports the `OMP_NUM_THREADS` variable correctly.

In pseudocode, the CPU test program looks like:

```
#pragma omp parallel
while (difftime(end,start) < interval)
        int a = 1 + 1;
```

This will create as many threads as the `OMP_NUM_THREADS` variable allows.

The results of running this through monitor are presented below.



As can be seen from the plots, the CPU usage is properly monitored when there are multiple cores being employed. A somewhat strange side effect is the increase in memory as the number of cores increases.

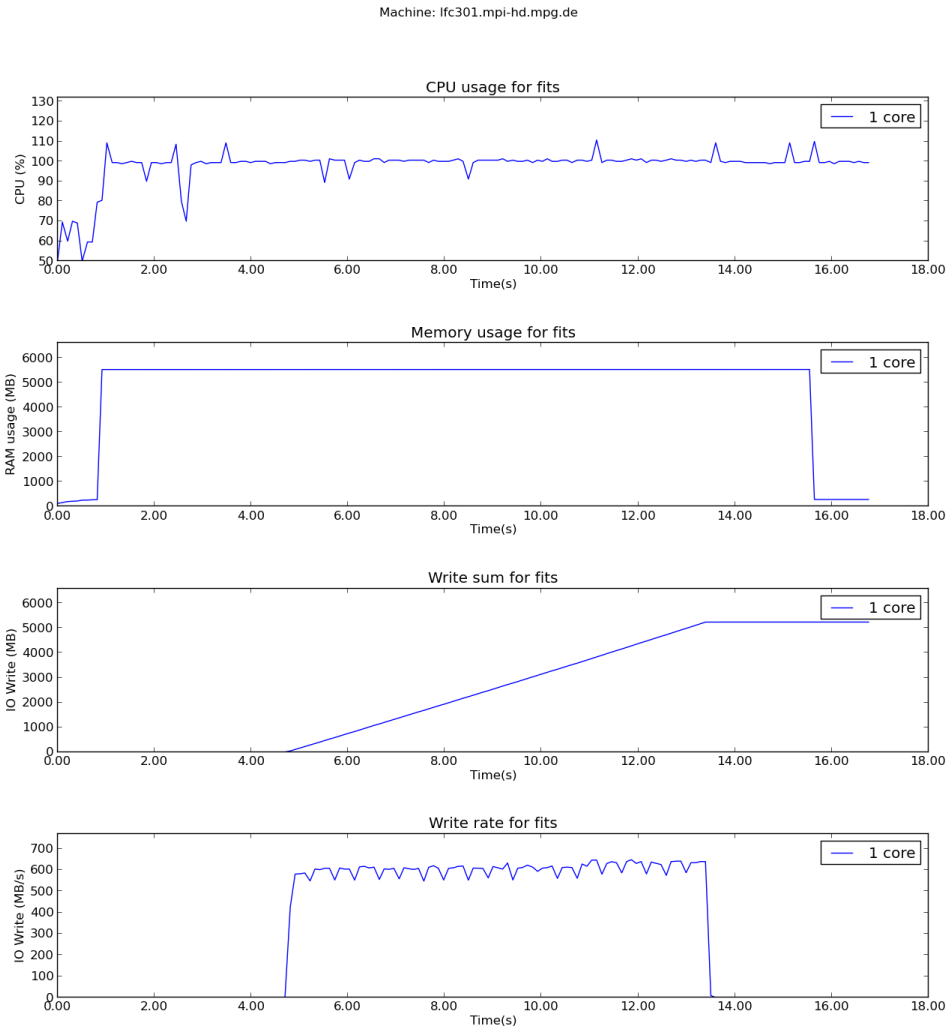
6.1.2 Disk I/O

In order to test for disk I/O, a simple Python testcase was developed that simply wrote a large file to disk. In pseudocode, it would look like:

```
def fits_gen(file_size):
    import numpy as np
    from astropy.io import fits

    data = np.zeros(file_size * 1024 ** 2 / 8)
    fits.writeto('data.fits', data=data, clobber=True)
```

The chosen filesize was 5 GB. Below, you can see the results of this measurement:



As can be seen from the plot, the cumulative write size is 5 GB and also the memory usage is about 5 GB, the extra coming from the libraries that need to be loaded. Therefore, the `Monitor` class and its methods yield accurate results when it comes to disk I/O.

6.2 ctobssim

6.2.1 Description

`ctobssim` is a function from the ctools suite. It is used to simulate astronomical observations given certain input parameters. For more details about the possible input parameters, see the official documentation.

The parallelism in `ctobssim` is realized at observation level. In short, this means that, in order to be able to see more than one CPU being used at a certain point in time, one needs to simulate more than one event. This is what we also did, as detailed in the next section.

In pseudocode, `ctobssim` looks like:

```
model = read_model("model.xml")
observations = read_observations("observations.xml")
event_lists = make_empty_lists(observations.size())

# omp parallel for
for (i = 0; i < observations.size(); i++)
    event_lists[i] = simulate_events(observation[i], model)

for (i = 0; i < observations.size(); i++)
        event_lists[i].save_to_file()
```

6.2.2 Input data

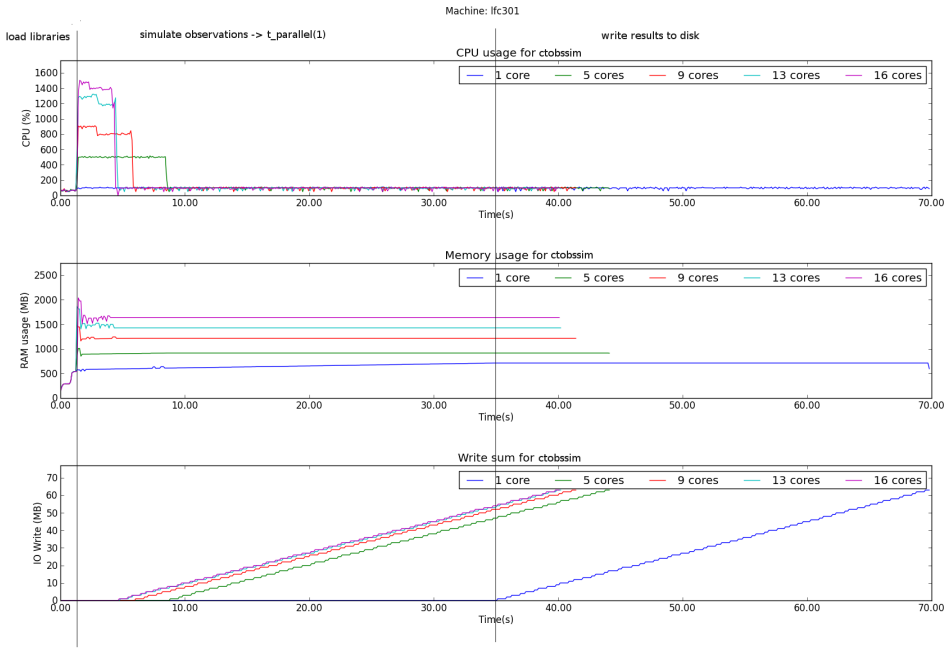
The input parameters for ctobssim were the following:

- **Astronomical model** Crab and Background
- **Calibration database** `$GAMMALIB/share/caldb/cta`
- **Instrument response function** `cta_dummy_irf`
- **RA** 83.63
- **DEC** 22.01
- **Duration of observations** 5000 s
- **Energy range** 0.1 - 100 TeV
- **Number of simulated runs** 100

6.2.3 Results

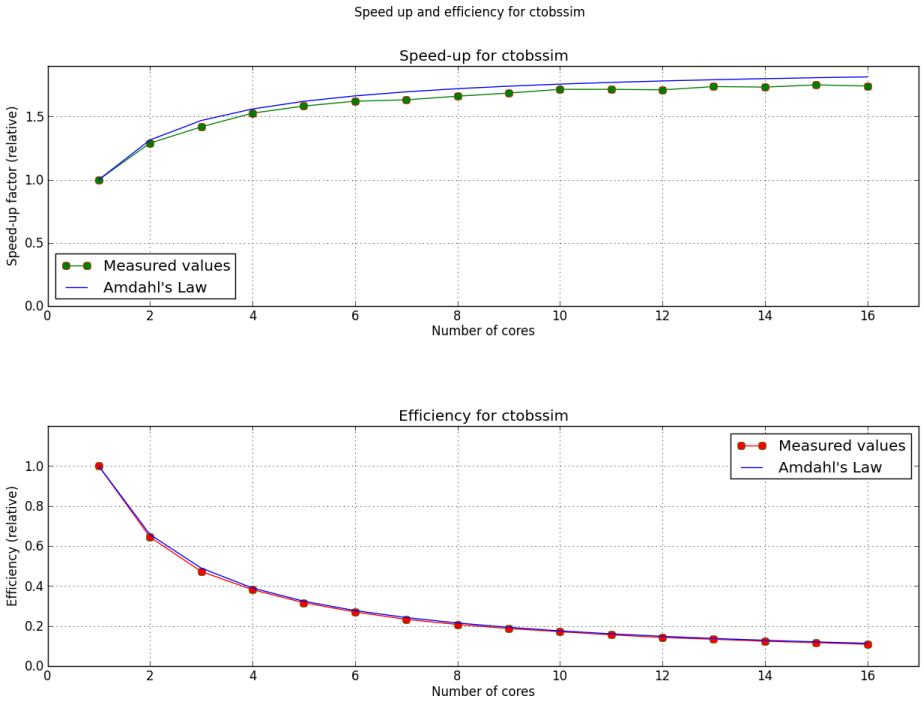
The results for the measurements of `ctobssim` are presented below.

The vertical lines on the plots represent the limits of the parallel portion of the code for the execution on 1 core. This `parallle_time(1)` was then plugged into Amdahl's Law in order to obtain the predicted speedup of the process.



The plots show that the serial part of the program and especially the part in which I/O is done is the most time consuming one. We can therefore assume that for a process with such long serial execution time, the speedup obtained from executing the process in parallel will not scale linearly.

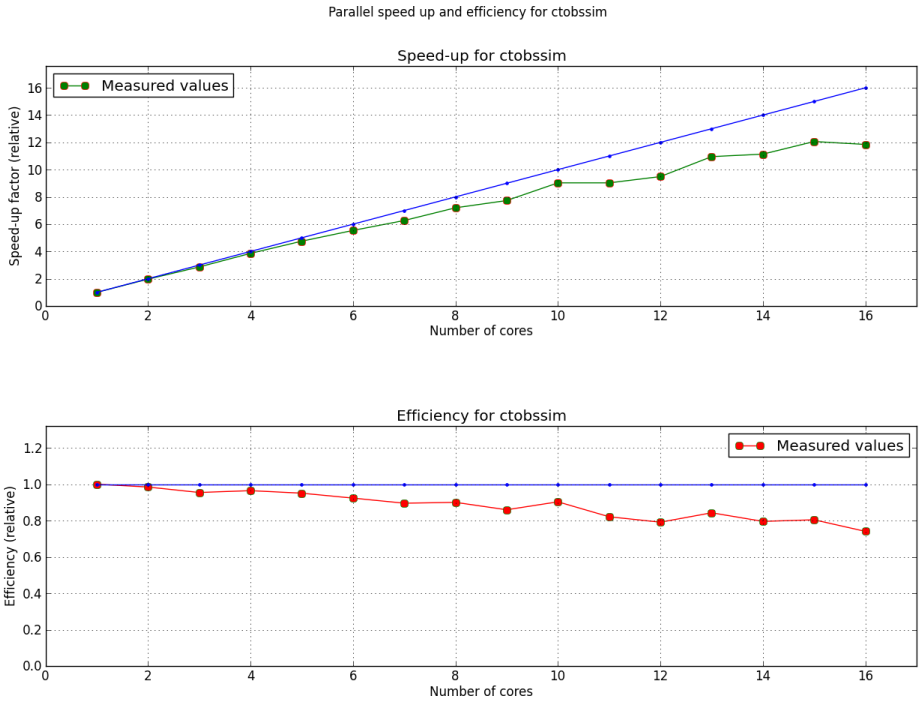
Below is a plot of the overall speedup of the process.



As can be seen, the execution of `ctobssim` respects Amdahl's Law almost prefectly.

A closer look to how the execution time of the parallel portions of the code scales with time reveals however that there

is probably some overhead that does not allow a linear scaling with time.



6.3 ctlike

6.3.1 Description

`ctlike` is another function from the ctools suite. It takes as input parameters a list of observations and it computes the maximum likelihood fit for that list of parameters. For more details about the possible input parameters, see the official ctlike

The parallelism in `ctlike` is implemented again at observation level.

In pseudocode, `ctlike` looks like:

```
model = read_model("model.xml")
observations = read_observations("observations.xml")
data = read_data("data.xml")
optimizer = make_optimizer(model, observations, data)
optimizer.run()
optimizer.model.save_to_file("fitted_model.xml")
void Optimizer::run(){
        // détails not important here ...
        // calls compute_likelihood() in a while loop
        // typically 100 to 1000 times until convergence is achieved
        while (..)
                this->guess_new_model_parameters()
                this->compute_likelihood()
}

void Optimizer::compute_likelihood(){
```

```
        likelihoods = std::vector<double>(this->observations.size())
        # omp parallel for
        for (i = 0; i < observations.size(); i++)
                // The per-observation likelihood is an expensive computation that
                // could be further parallelized ... but details are not important here
                likelihoods[i] = compute_likelihood_for_one_observation(i)
        self->likelihood = likelihoods.sum()
}
```

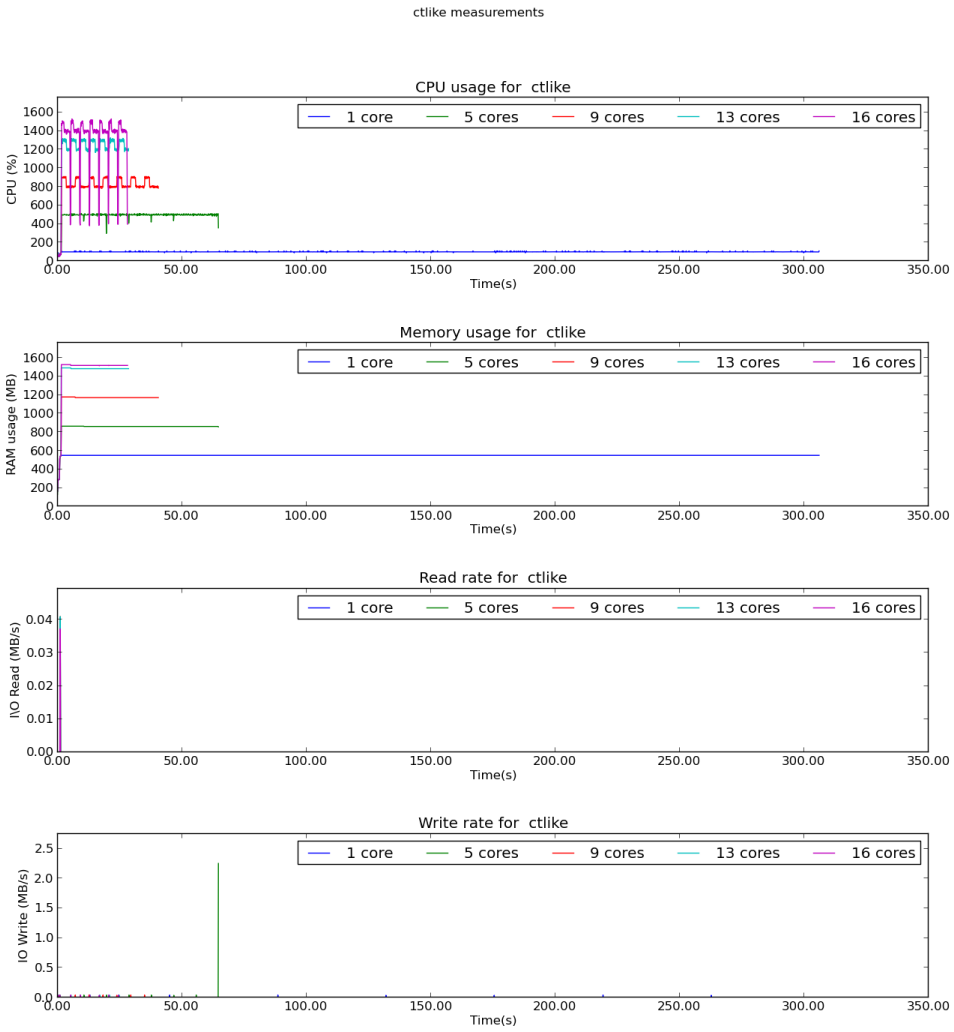
6.3.2 Input data

- **Astronomical model** Crab and Background
- **Source of data** ctobssim simulated runs
- **Events/run** 360
- **Analysis type** unbinned
- **Energy range** 0.1 - 100 TeV
- **Number of evaluated runs** 100

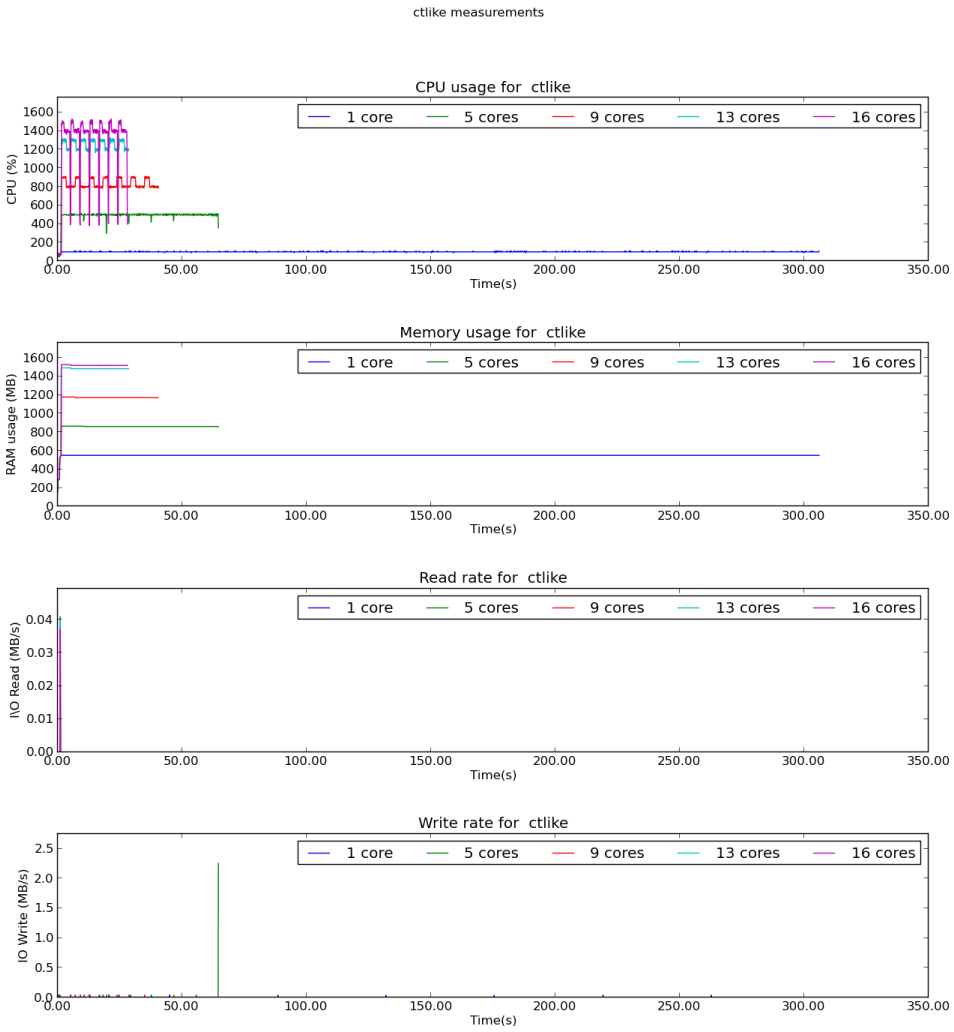
6.3.3 Results

The results for `ctline` are presented below.

As can be seen, even though `ctlike` is allowed to use for example 16 cores, it will only 14 or 15 cores for the simulation. This behaviour has not yet been properly explained.



As was expected, `ctlike` is doing almost no I/O at all. Another important point to note is that the biggest part of `ctlike` is parallelized so we can expect good speedups when using multiple cores.



The measured speedup does not conform with the value predicted by Amdahl's Law. However, it can be seen that the execution time is greatly reduced when using multiple cores. Further investigation is needed to see why the OpenMP parallelization fails to yield an ideal efficiency.

6.4 pipeline analysis

6.4.1 Description

The pipeline analysis consists of different that are run consecutively. Thus, the events are simulated by ctobssim, they are filled into enegy bins using ctbin and the maximum likelihood fit is computed using ctlike.

The advantage of the pipeline is that it can run these events one after the other instead of separating the processes from one another. Another advantage is that the data can be stored in memory thus relinquishing the need to physically save the results to disk.

As before, parallel behaviour can be observed only when running the pipeline for more than one event. The parallelized portions of the code lie in `ctobssim` and `ctlike` while ctbin contains only serial code.

6.4.2 Input data

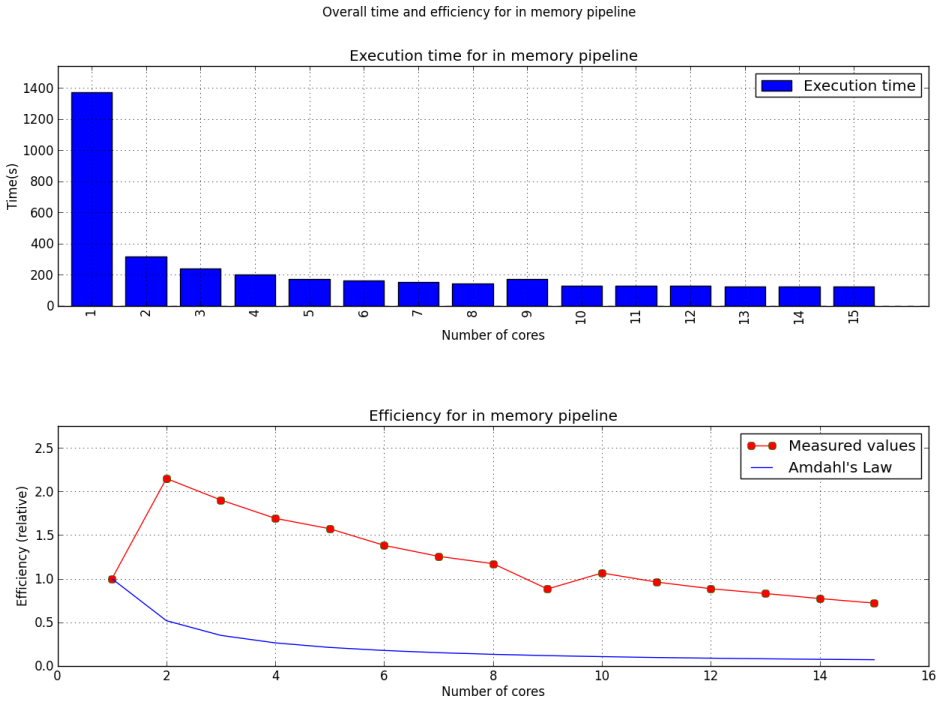
The pipeline was run with the following input data:

- **Astronomical model** Crab and Background
- **Calibration database** `$GAMMALIB/share/caldb/cta`
- **Instrument response function** `cta_dummy_irf`
- **Duration of observations** 5000 s
- **Energy range** 0.1 - 100 TeV
- **Number of simulated runs** 100

6.4.3 Results

The results of the in memory pipeline can be seen below.

As can be seen, the execution on a single core takes a lot more time than it does on multiple cores. This has a strong effect on the efficiency as can be seen in the next plot.



The multiple core execution time is much shorter than the single core time. This will lead to a measured efficiency greater than 1! It was not yet determined why this happens. Even though the pipeline analysis was run multiple times, it yielded similar results each time.

CHAPTER

SEVEN

FUTURE DIRECTIONS

7.1 Measuring FITS I/O

7.1.1 Goal

Measure the FITS I/O speed for CTA format event lists and compare to ROOT and HDF5.

For FITS look at the speed of:

- CFITSIO
- CFITSIO Python wrapper
- astropy.io.fits
- astropy.io.table
- GammaLib (e.g. write with *ctobssim* and read / write with *ctselect*)

7.1.2 Methods

Define test datasets of different sizes (1 MB to 10 GB with ~ 10 values and log spacing).

Do operations so that I/O speed dominates execution time:

- Event selection using box cuts
- Histogram or simply sum data

These days memory is pretty large (10 GB to 100 GB are typical), maybe we should also measure memory to CPU I/O bound operations?

7.1.3 Some references

- http://www.hdfgroup.org/HDF5/RD100-2002/HDF5_Performance.pdf
- http://heasarc.gsfc.nasa.gov/fitsio/c/c_user/node119.html
- http://adsabs.harvard.edu/abs/2009CoPhC.180.2499A
- http://adsabs.harvard.edu/abs/2011JPhCS.331c2010A

CHAPTER

EIGHT

LINKS

The following is a link collection that contains different links that were useful for this project.

- Timing tools:

  http://en.wikipedia.org/wiki/Time_(Unix)

- Profiling tools:

  http://software.intel.com/en-us/intel-parallel-studio-xe

  http://en.wikipedia.org/wiki/LTTng

  http://code.google.com/p/gperftools/

  https://wiki.engr.illinois.edu/download/attachments/114688007/amplifier_xe_linux.pdf

- Tutorials:

  http://www.ibm.com/developerworks/library/l-gnuprof.html

  http://en.wikipedia.org/wiki/Performance_analysis

  http://en.wikipedia.org/wiki/Profiling_(computer_programming)

  http://nbviewer.ipython.org/urls/raw.github.com/jrjohansson/scientific-python-lectures/master/Lecture-6B-HPC.ipynb

  http://www.softeng.rl.ac.uk/media/uploads/publications/2010/07/ProfilingTutorial.pdf

- Articles:

  http://www.gotw.ca/publications/concurrency-ddj.htm

  http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck/

  http://web.mit.edu/newsoffice/topic/the-multicore-future.html

  http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

- Fitting:

  http://jarrodmillman.com/scipy2011/pdfs/brefsdal.pdf

  http://adsabs.harvard.edu/abs/1979ApJ...228..939C

- General links:

  http://www.debugging-guide.com/

  http://spiff.rit.edu/classes/phys445/lectures/radec/radec.html

- Google Performance Tools:

  http://code.google.com/p/gperftools/?redir=1

- Simple Python Profiling tutorial:

  http://nematodes.org/martin/teaching/programming-topics/profiling-python-for-cpu-and-memory-usage/

- Online courses

  http://www-users.york.ac.uk/~mijp1/teaching/4th_year_HPC/notes.shtml

  http://www.citutor.org

- Books:

  Quinn, "Parallel Programming in C with MPI and OpenMP", Amazon

  Grötker, "Developer's Guide to debugging", Amazon