Timeplotters:
two tools for visualizing temporal data

Eugene Kirpichov <ekirpichov@gmail.com>

June 4, 2012

Contents

1 Introduction

splot and **timeplot** are tools for visualizing temporal data, especially useful for visualizing data from program logs.

- **splot** draws a single Gantt-like chart with a birds-eye view of the activity of a number of concurrent processes, each of which may be in one of several different states at each moment (e.g. processing one of several jobs, or being in a particular stage of processing). This allows to see peculiar system-level behavior patterns and usually allows to instantly isolate system-level performance bottlenecks which very often show themselves as distinct visual patterns. See section 3.1 for a very characteristic example of what non-trivial aspects of a program's behavior **splot** can show.
- **timeplot** can draw quantitative graphs of several values at once, e.g. you can use it to compare the distribution of database access latencies from two machines; or to draw the number of requests being concurrently processed by each server at each moment.

The main design goal is to make it possible to use the tools for exploratory analysis and visually answer most questions about your real-world logs with shell one-liners, without modifying the original program. This implies several characteristics:

- Input is trivial to generate from raw logs by usual text processing tools such as awk or perl
- The same input data can be used to generate different graphs
- Performance is good enough to draw graphs over many millions of events in reasonable time

This manual is structured as follows: after an introduction, we describe first **splot** and then **timeplot**, describing each with the following structure:

- Concepts — the definitions and examples necessary to understand the format and logical structure of the input
- Input format — syntax and semantics of the input
- Main features and examples of their usage
- A complete description of all options
- A gallery of real-life plots made with the tool

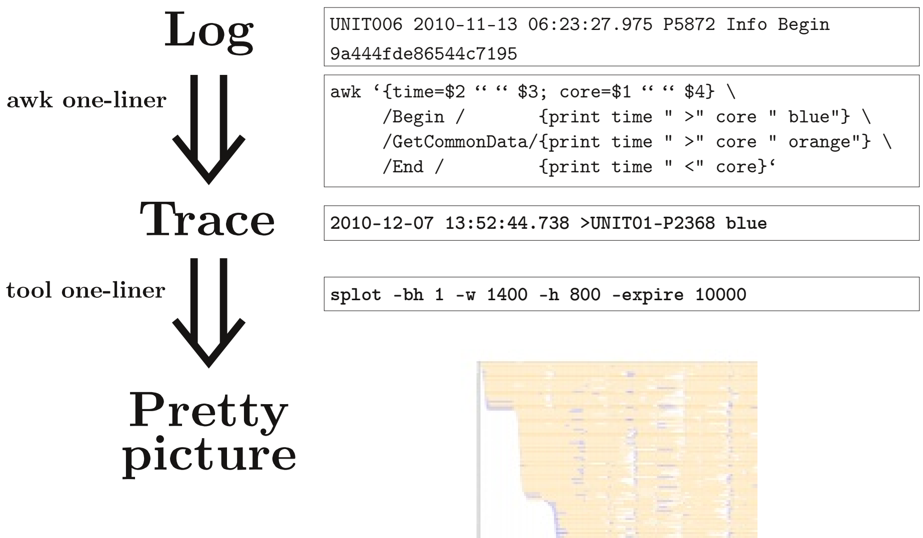

Figure 1: The general pattern of usage of both **timeplot** and **splot**.

2 General usage

Both **timeplot** and **splot** accept input in a strictly specified format, not arbitrary logs. A file in this format is called *trace*. However, this format is designed to be trivial to generate from log files using text processing tools such as awk, sed or perl.

Note: I found **awk** to be the most useful tool of these. I know that some people have used perl or python to generate traces, but I don't know enough of perl and find Python not terse enough for writing one-liners, so I always use **awk**.

Note: You will immensely benefit from learning to efficiently use **awk**. If you are not yet an **awk** guru, please proceed to read one of the multitude of available tutorials. Concentrate on googling for "awk one-liners explained" or something like that: since you are going to use the tools interactively, it is vital to be able to write **awk** one-liners for typical tasks. However, in most cases it will suffice to know just the very basics of **awk** explained on the first page of any decent tutorial.

The general pattern of usage is displayed on figure 1.

This is how it looks in the shell: you use an awk one-liner to generate the *trace* and invoke the tool on it.

```
$ awk '/some log entry/{emit an event into trace} \
       /another log entry/{emit another event...}' \
```

```
       log.txt > trace.txt
$ splot -if trace.txt -o picture.png ..options..
```

This is usually easy to type right in the shell. You can make it into a one-liner (long but still suitable for interactive usage) like this:

```
$ awk '...' log.txt > trace.txt && splot ...
```

Interactive usage is also eased by the fact that you usually invoke this one-liner repeatedly with small changes (playing with the tools' options, or selecting some or other events from the logs).

Let us look at a real-life example without considering it in too much detail. Use it only for the purpose of understanding the general pattern of usage.

In this example, we're drawing the activity of a computational cluster using **splot**. There's a bunch of worker processes which collaboratively process independent tasks from a shared queue. Every task has two stages: fetch some data from memcached and then do the actual computations.

The log to this picture has been unfortunately lost, but it had entries that looked like this:

```
UNIT006 2010-11-13 06:23:27.975 P5872 Info Begin 9a444fde86544c7195
```

The meaning of the entries was as follows:

- There's machine name, timestamp and process ID in the entry. The last field is the task ID which is irrelevant for our current drawing purposes.
- `Begin` means "starting to execute a task".
- `GetCommonData` means "finished getting the task's data from memcached, starting actual computations".
- `End` means "computations for a task finished".

```
awk '{time=$2 " " $3; core=$1 " " $4} \
     /Begin /       {print time " >" core " blue"} \
     /GetCommonData/{print time " >" core " orange"} \
     /End /         {print time " <" core}' log.txt > trace.txt
splot -if trace.txt -o splot.png -bh 1 -w 1400 -h 800 -expire 10000
```

The result is shown on figure 2.

This example will be further considered in detail in section 3.1.

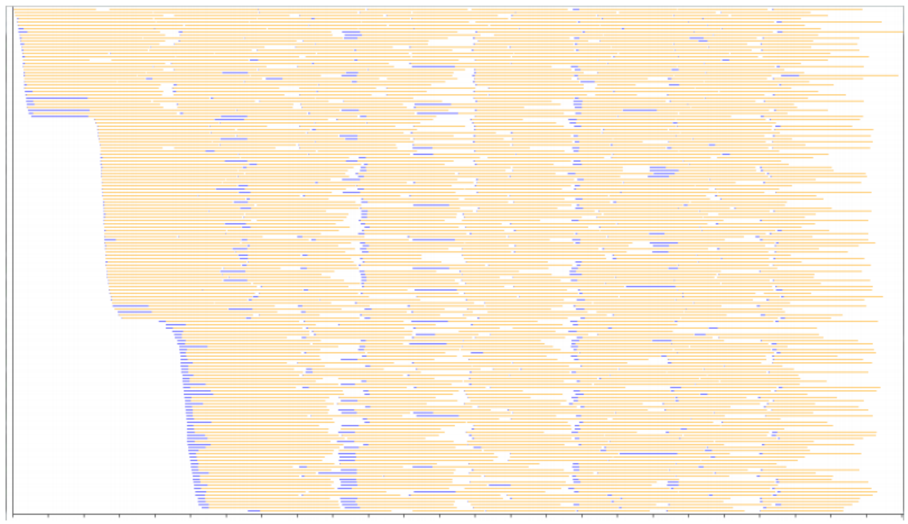

Figure 2: Motivating example for splot

3 splot — visualizing behavior of many concurrent processes

splot draws a single Gantt-like chart with a birds-eye view of the activity of a number of concurrent processes, each of which may be in one of several different states at each moment (e.g. processing one of several jobs, or being in a particular stage of processing).

The input to **splot**, in the basic form, is a sequence of events: *timestamp T: process X started activity Y*, *timestamp T: process X finished activity Y* (where X is a process identifier — an arbitrary string, and Y is a color). **splot** then draws a graph where the horizontal axis is time and the vertical axis is process ID (by somehow mapping the arbitrary string ID to a position on the vertical axis — currently by time of the first event).

3.1 Motivation

In this section we shall consider the plot from figure 2 and see how it helps highlight a number of problems in the program whose behavior was visualized, that would be difficult to find without **splot**.

Recall that here, the X axis is time, the Y axis is cluster worker (core), orange is computations and blue is fetching data from memcached.

There are several anomalies on this picture:

Whitespace on the left: Slopes. The beginning of the picture corresponds to the program startup — the first tasks are being fed into the shared queue and workers pick them up and start executing. The whitespace means that a large fraction of the program execution time is spent just warming up, while the workers essentially do nothing. The slopes mean that workers do not receive the first tasks instantly. Either it takes time to pick them from the queue, or they are being fed to the queue not quickly enough.

Whitespace on the left: Plateaus. The plateaus mean that there are moments when no tasks are being picked up at all. Either the queue is hung up, or the task producer. We also see how big an impact this has on the overall cluster utilization: a single 1-second plateau is worth 160 seconds of computations (as there are 160 workers on this picture)!

The slope on the left is nonlinear. The slope becomes less vertical at the end of each slope line. This means that task pickup rate (or perhaps task generation rate) is not constant. There's a "tail" in task delivery times.

Some blue bars are rather long. This means that memcached fetches sometimes take a long time — comparable with computation time. We should optimize them.

Vertical patterns of white space and blue bars lining up on their left edge. This means that there are moments when everyone's got nothing to do and the task queue is empty, and then suddenly a lot of tasks appear in the queue and everyone is busy again. This is probably a problem with the task producer — maybe it is sending tasks in batches, or something like that.

A lot of white space on the right. This means that in the end of the program execution, when all tasks are already in the queue and no new ones will appear, a lot of time is spend when faster workers wait for slower workers. The program finishes when the very last worker finishes. The fraction of this whitespace is very well worth 5-10% of the total program time. This anomaly is called "the long tail effect" and can be eased by increasing the task granularity (i.e. submitting many short tasks rather than several long ones). However, this will obviously increase load on the queue, so we have a trade-off here.

We see that a simple picture of the cluster behavior showed us quite a few non-trivial things. We leave it to the reader's imagination to think how many of these anomalies could have been found without a visualization, by staring or quantitative analysis of the logs.

3.2 Concepts

Let us now consider in detail all the concepts necessary to understand and use **splot**.

Track (also process) An entity which changes its state between several values (performs several activities) over time. At any moment, a process is doing at most one activity. We're usually interested in seeing the visual pattern emerging between different tracks. For example, it is often meaningful to assign 1 track = 1 thread in a multi-threaded program (in a distributed program one should of course include the machine into the track id).

Activity A period during which a process is in a particular problem-specific logical state. A single activity is drawn with a single color, as a bar which is horizontally as long as the activity (if there are a lot of tracks, this bar can be as thin as a hairline). For example, if we have a program whose threads are either computing, doing IO or waiting idly, we may depict the 'computing' activity with orange color, IO as blue and idle waiting as lack of activity (no color at all).

Event Mark of the beginning or end of an activity on a particular track, specifying the activity's color. There are also "text" events which allow to draw text markers above the usual colored bars. An event has a timestamp, track and event type (activity start / activity end / text). For example, when our hypothetical program starts doing IO in thread T, we should represent this in **splot**'s input as an event "start blue activity on track T".

Color Colors in **splot** can be specified in several ways. In the simplest form, it can be an SVG color name or hex code (e.g. "red" or "lightblue" or "#ff0033"), it can be an arbitrary string (then a random color will be generated so that different strings correspond to different colors — this is useful to color-code an unknown number of different types of activities, e.g. if you have worker processes servicing several clients and you wish to get a picture of who services whom and color-code the clients: "client-5"), and it can be an arbitrary string within a color scheme (e.g. "/success/client-5" or "/failure/client-5", and you might define the "success" colorscheme to consist of several greenish colors and "failure" of reddish).

Color scheme A list of colors which will be cycled between when generating colors for tracks whose color is not specified explicitly by a SVG color name or hex code.

3.3 Input format

The input to **splot** consists of a series of events.

Format	Meaning
TIME >TRK COL	Start activity of color COLOR on track TRK at time TIME (if there already is an activity, finish it and start a new one instead).
2010-10-21 16:45:09,431 >r2b3.t5 blue	
TIME <TRK	Finish the current activity of track TRK at time TIME
2010-10-21 16:45:10,322 <r23.t5	
TIME <TRK COL	Finish the current activity of track TRK at time TIME, overriding the color it was given at the start by COLOR. This is useful, e.g. do indicate that an activity failed by drawing it with red color (obviously when the activity begins, we don't know if it will fail or not).
2010-10-21 16:45:10,322 <r2b3.t5 red	
TIME !TRK COL TXT	Draw text TXT with color COL on track TRK, left-justified at time TIME
2010-10-21 16:45:10,322 !r2b3.t5 black read()	

3.4 Simple example

This example has two tracks "thread-1" and "thread-2" and demonstrates most of the basic features.

```
2010-10-21 16:45:12.014 >thread-2 blue
2010-10-21 16:45:13.329 >thread-1 blue
2010-10-21 16:45:13.635 <thread-1
2010-10-21 16:45:13.800 <thread-2
2010-10-21 16:45:13.810 >thread-1 blue
2010-10-21 16:45:13.810 >thread-2 orange
2010-10-21 16:45:14.010 >thread-1 orange
2010-10-21 16:45:14.258 <thread-2
2010-10-21 16:45:14.623 <thread-1
2010-10-21 16:45:14.629 >thread-2 orange
2010-10-21 16:45:15.138 >thread-1 orange
2010-10-21 16:45:15.319 >thread-1 blue
2010-10-21 16:45:15.502 <thread-1 red
2010-10-21 16:45:15.502 !thread-1 black HTTP 500
2010-10-21 16:45:16.112 <thread-2
```

Invoke **splot**:

```
$ splot -if splot.simplest-example.trace \
       -o splot.simplest-example.png -w 400 -h 160
```

Result:

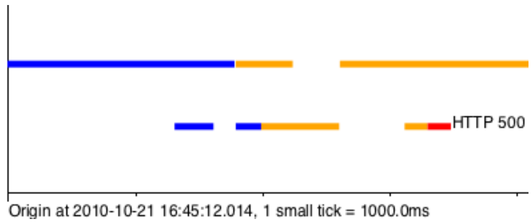

Figure 3: A trivial example of **splot** usage

3.5 Advanced features

3.5.1 Bar height

Bar height is specified with `-bh`: either `-bh fill` or `-bh HEIGHT` (e.g. `-bh 5`). `fill` means "set bar height to fill the whole vertical space". Consider the previous simple example, here it is with `-bh fill`:

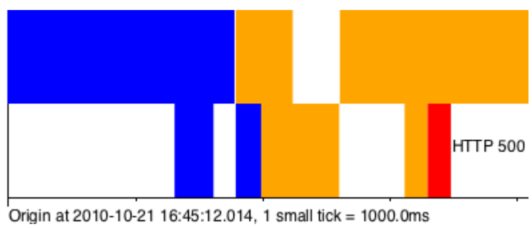

Figure 4: A trivial example of **splot** usage with the `-bh fill` option

`-bh fill` is vital when you have a lot of tracks (at least dozens). Perhaps it should be the default.

3.5.2 Expiring activities — if "<" is missing

In systems where components can crash (which is, most systems :)), it might happen so that your log catches only the beginning of an activity but not its end, because the component has crashed in the middle. You can tell **splot** "expire all activities if they take longer than X seconds" by using `-expire X`; then, if an activity has not finished within X seconds, **splot** will draw a dashed line and an X marker, meaning that the process probably crashed somewhere on the dashed line[1].

Consider a small example similar to the one we had before.

[1]It's probably meaningful to specify expiration times for each activity separately, but currently **splot** has just a single global option

```
2010-10-21 16:45:12.014 >worker-2 blue
2010-10-21 16:45:13.329 >worker-1 blue
2010-10-21 16:45:13.635 <worker-1
2010-10-21 16:45:13.800 <worker-2
2010-10-21 16:45:13.810 >worker-1 blue
2010-10-21 16:45:13.810 >worker-2 orange
2010-10-21 16:45:14.010 >worker-1 orange
2010-10-21 16:45:14.258 <worker-2
2010-10-21 16:45:14.623 <worker-1
2010-10-21 16:45:14.629 >worker-2 orange
2010-10-21 16:45:15.138 >worker-1 orange
2010-10-21 16:45:15.319 >worker-1 blue
2010-10-21 16:45:16.512 <worker-2
2010-10-21 16:45:18.412 >worker-2 blue
2010-10-21 16:45:20.112 <worker-2
```

Let us draw it with an expiration time of 2000 milliseconds:

```
$ splot -if splot.expire-example.trace -o splot-expire-example.png \
        -expire 2000 -bh 15 -w 400 -h 160
```

Here's what we get:

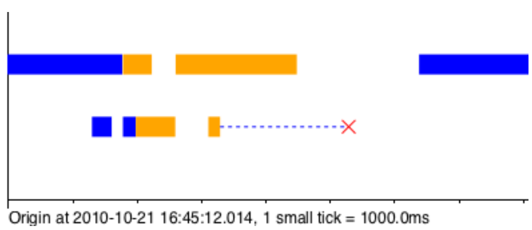

Figure 5: Using the `-expire` option with **splot**

Here we see that perhaps "worker-1" (here drawn second, as its first event happened later than worker-2's) crashed and that's why it didn't do anything in the later parts of the graph.

Figure 6 shows a more complex real-life example from a cluster (the input log has unfortunately been lost). Here a large number of workers are preempted at once.

3.5.3 Phantom color — if ">" is missing

Sometimes you process logs which start in the middle of the program's execution, so the log doesn't catch the beginning events of activities that were active at the

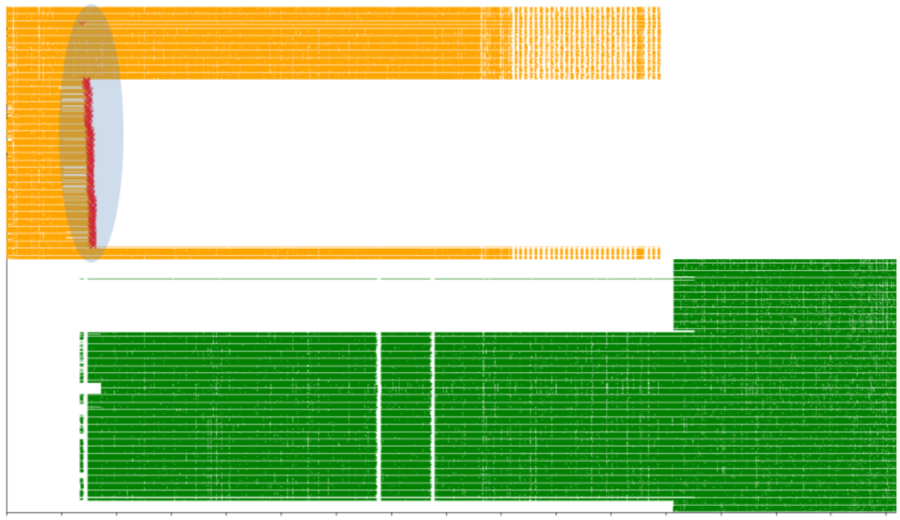

Figure 6: Using the `-expire` option with **splot**: a real-life example

moment the log was started (however, the log *does* catch their finishing events). **splot** can display such "phantom" activities in a color of your choice, using the `-phantom COLOR` option. Specifically, if a track starts with a "<" event instead of ">", then **splot** will assume that there was a ">" event with color COLOR in the past on this track.

Consider the same example as above, but let us cut it in the beginning, as if we had a truncated log:

```
2010-10-21 16:45:14.010 >worker-1 orange
2010-10-21 16:45:14.258 <worker-2
2010-10-21 16:45:14.623 <worker-1
2010-10-21 16:45:14.629 >worker-2 orange
2010-10-21 16:45:15.138 >worker-1 orange
2010-10-21 16:45:15.319 >worker-1 blue
2010-10-21 16:45:16.512 <worker-2
2010-10-21 16:45:18.412 >worker-2 blue
2010-10-21 16:45:20.112 <worker-2
```

Now invoke **splot**:

```
$ splot -if splot-phantom-simple-example.trace \
        -o splot-phantom-simple-example.png \
        -phantom gray -w 400 -h 160 -bh 5
```

And here's what we get:

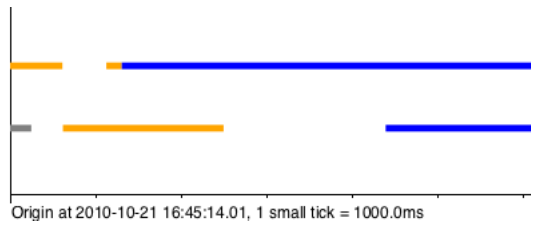

Figure 7: Using the `-phantom` option with **splot**

We see a gray bar in the beginning of worker-2's track. This is because the first event on this track was `2010-10-21 16:45:14.258 <worker-2`, i.e. an activity closing event, which means that the opening event was missed.

3.5.4 Color auto-generation

How do we color-code activities in **splot**'s input if the set of possible activity types is not known in advance? E.g. you have a set a of worker processes which service a fairly small but unknown number of clients. You assign tracks to worker processes, and you wish to color-code the clients to see a picture of who services whom and when.
In this case, you can simply use the client's id as color: for colors that do not parse as SVG color names or hex codes, **splot** will generate a random color from a default contrast color scheme.
Let us consider a real-life example. In this example, again some workers are processing tasks, but from different clients. The trace looks like this:

```
...
2011-07-27 02:42:03.485 <RACK5UNIT067.37dc
2011-07-27 02:42:03.492 >RACK2UNIT067.4ca8 8610
2011-07-27 02:42:03.495 >RACK4UNIT075.5b15 8610
2011-07-27 02:42:03.496 >RACK4UNIT067.ec72 877B
2011-07-27 02:42:03.496 >RACK4UNIT067.f3fe 877B
2011-07-27 02:42:03.496 >RACK5UNIT017.0c21 8610
2011-07-27 02:42:03.498 >RACK2UNIT030.9e7a 071C
...
```

So, we use track names of the form `MACHINE.WORKERID` and instead of using color at ">" events, we use the client ID, asking **splot** to color-code clients for us. The result is shown on figure 8. Here red color denotes tasks that completed unsuccessfully and other colors (e.g. green and blue; red is excluded from the default color scheme precisely for situations like this) encode different clients[2].

[2] Yes, as this rather crazy picture might hint, the program *did* have lots of problems — exposing them, that's what **splot** is for.

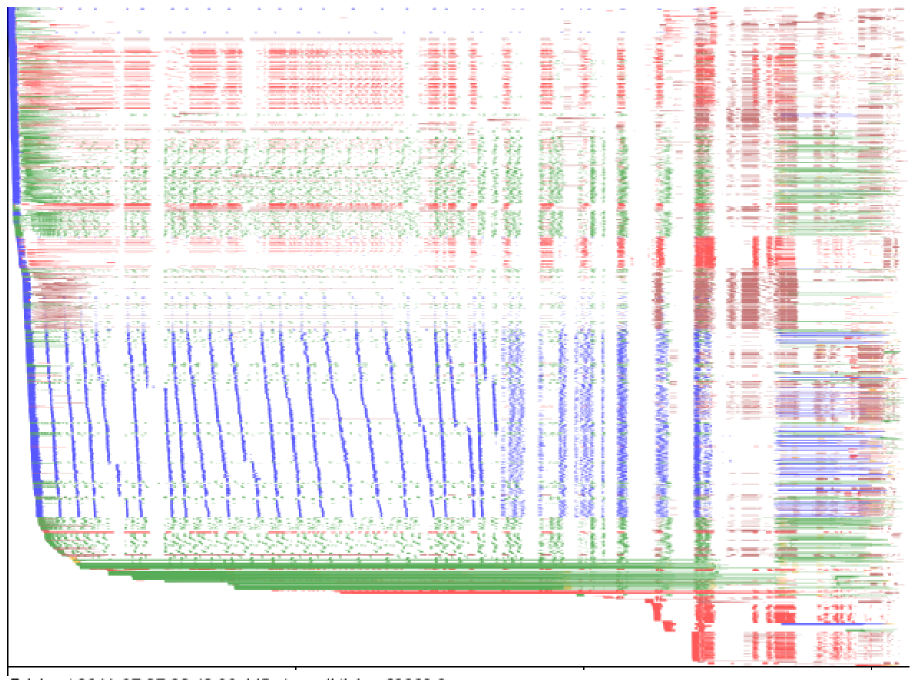

Figure 8: Automatic color generation with **splot**

Assume the same case as above with worker processes servicing different clients. Assume also that worker processes might be in two modes: regular and workstealing (if they have nothing to do with their current client, they try to service tasks of some other client). We wish to depict regular tasks and tasks picked by work-stealing in different shades: e.g. regular tasks as bright colors and work-stolen ones as pale. We still wish to use color generation for both. This can be achieved by using color schemes. Specifically, we'll color-core regular tasks by `TIMESTAMP >WORKER CLIENT` and work-stolen tasks by `TIMESTAMP <WORKER /ws/CLIENT`. This tells **splot** to generate colors for regular tasks from the default color scheme and for work-stolen tasks from the `ws` color scheme, which we must specify in `-colorscheme` parameters, e.g. `-colorscheme ws='lightblue lightgray lightgreen pink beige'` (use more colors if you wish to distinguish between more clients).

3.6 Option reference

Option	Meaning	Default value
`-if INFILE`	Input filename	*Required*
`-o PNGFILE`	Output filename	*Required*

Option	Meaning	Default value
`-w WIDTH`	Output width, pixels	`640`
`-h HEIGHT`	Output height, pixels	`480`
`-bh BARHEIGHT`	Vertical height of each track's activity bars, pixels, or 'fill' to use all the available vertical space	`fill`
`-tf PATTERN`	Format of time in the input file as in http://linux.die.net/man/3/strptime but with fractional seconds supported via `\%OS` - will parse `12.4039` or `12,4039`. Also, `%^[+-][N]s` will parse seconds since the epoch, for example `%^-3s` are milliseconds since the epoch (N can only be 1 digit)	`%Y-%m-%d %H:%M:%OS`
`-tickInterval MILLIS`	Ticks on the X axis will be this often	`1000`
`-largeTickFreq N`	Every N'th tick will be larger than the others	`10`
`-sort SORT`	Sort tracks by time of first event (`-sort time`) or by track name (`-sort name`) — see "track sorting" above	`name`
`-expire MILLIS`	Expire activities that do not finish within MILLIS milliseconds — see "expiring activities" above	*none (don't expire)*
`-phantom COLOR`	Set the phantom color which is used if the first event on a track is "<" — see "phantom color" above	*none (no phantom color)*
`-fromTime TIME`	Clip the picture on the left (time in the format of `-tf`, i.e. same as in the input)	*none (don't clip)*
`-toTime TIME`	Clip the picture on the right (time in the format of `-tf`, i.e. same as in the input)	*none (don't clip)*
`-numTracks N`	Explicitly specify the number of tracks for better performance on very large data (see section "Performance" below)	*none (compute from input)*
`-colorscheme SCHEME COLORS`	Declare a colorscheme (see "Color schemes" above). Can be used multiple times. Scheme is an arbitrary string, e.g. `pale` or `bright`. COLORS is a space-separated list of colors in SVG or hex, e.g. `'red green 0x0000FF'`	*none*

3.7 Gallery

Figures 9–19 show a number of pictures produced by **splot** in various real-life situations. Most of them look really creepy and expose different kinds of performance problems in the programs whose behavior they depict. The author already does not remember the precise reasons for the problems — think of it as a horror museum exposition and use your imagination.

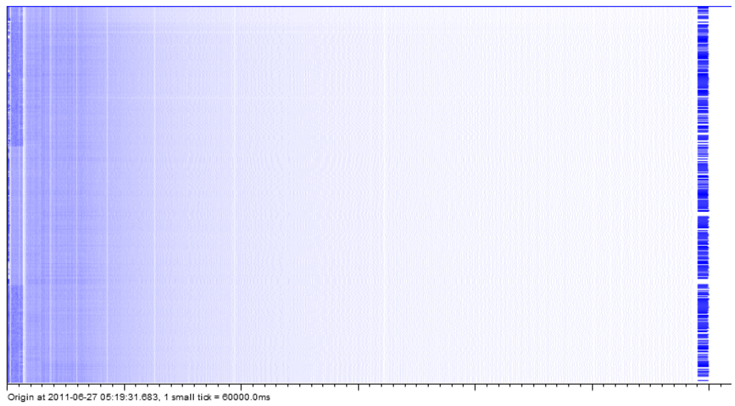

Figure 9: Blue: working, white: waiting. The task queue's performance was gradually becoming worse and worse.

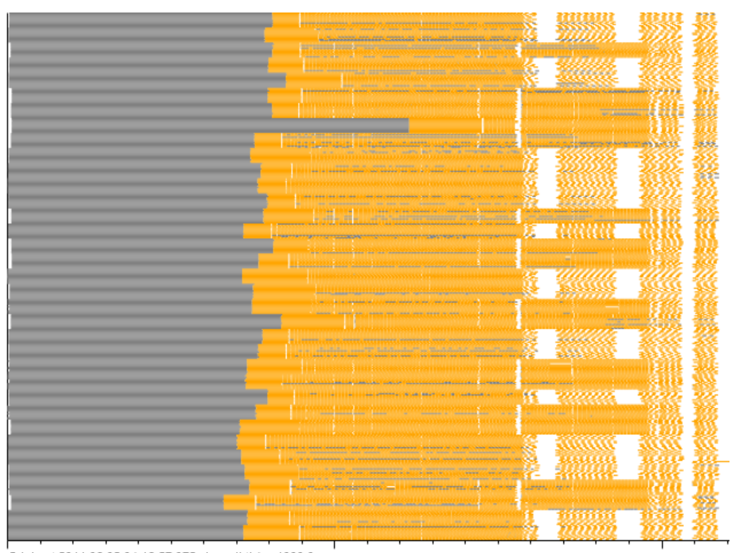

Figure 10: Orange: working, gray: starting, white: waiting. The first time a program loads, it takes a long time. Loads on the same machine take the same time — .NET DLL caching (here the sort-by-track-name option is used, currently absent until reimplemented).

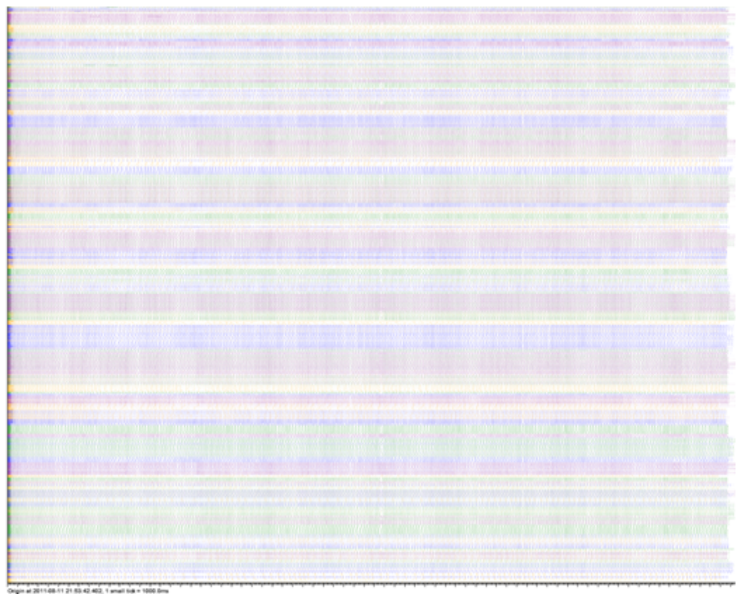

Figure 11: Colors encode which shard of a task queue was being used. Initially the shards are fast, then they run slower but smoothly. Everything's fine.

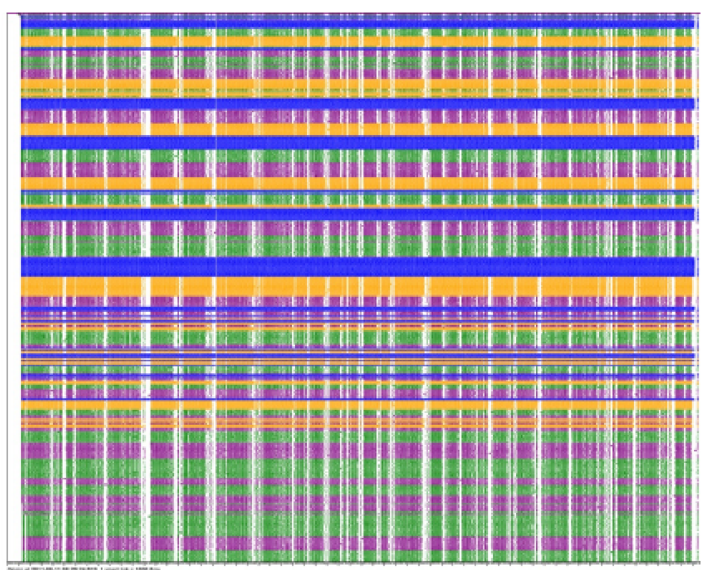

Figure 12: Colors encode which shard of a task queue was being used. Green and purple have problems, yellow and orange don't. Turned out green and purple corresponded to the same physical queue server which had to sustain double load.

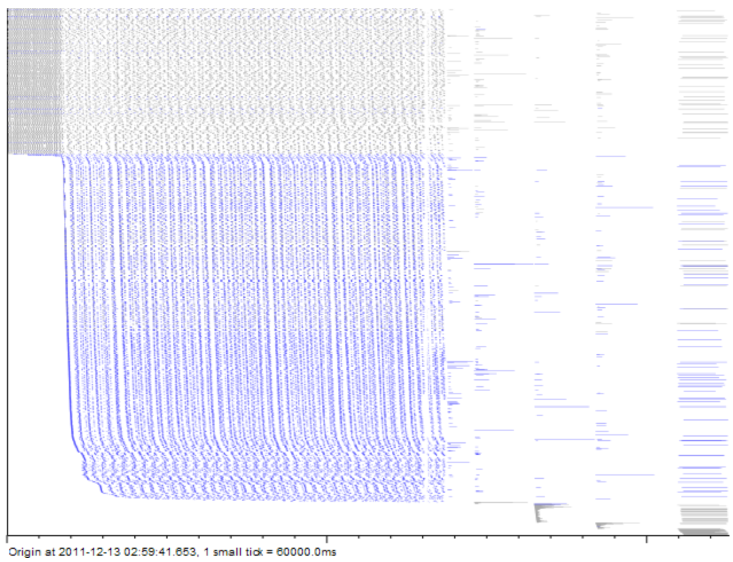

Figure 13: Gray: normal tasks, blue: tasks picked by workstealing. After some period, workstealing begins and many workers start processing the job — slowing down the queue, but total throughput is higher.

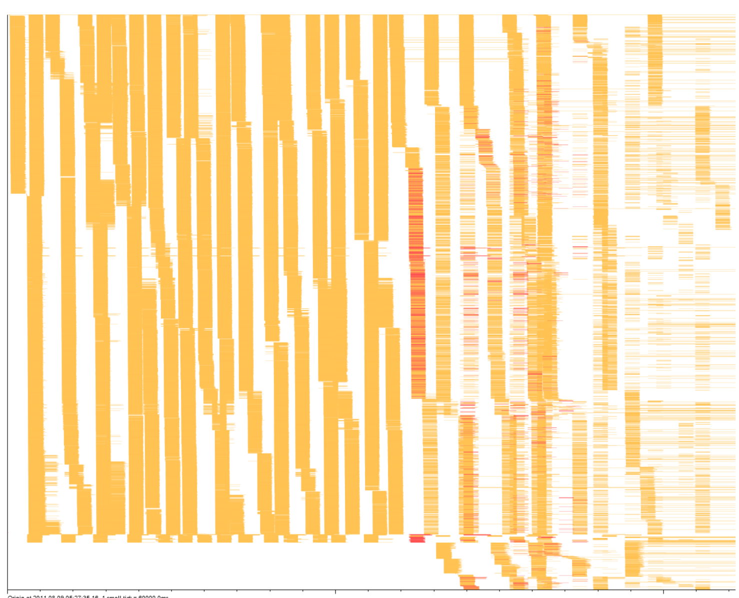

Figure 14: Orange: working, red: preempted and lost work. The task queue prefetch feature was broken, leading to very strange patterns of task queue utilization.

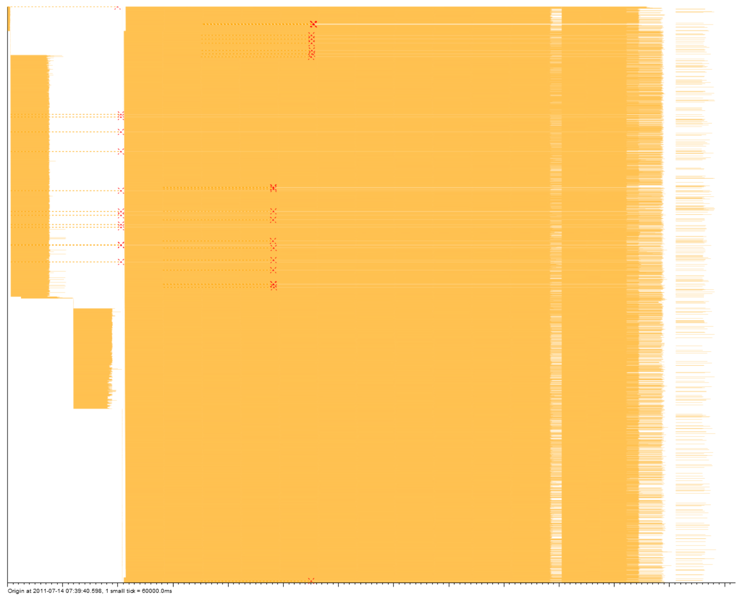

Figure 15: A simple 2-stage job: a couple of insufficiently parallel data preparation steps, then a long stream of tasks utilizing the cluster well. Two worker machines died in the process.

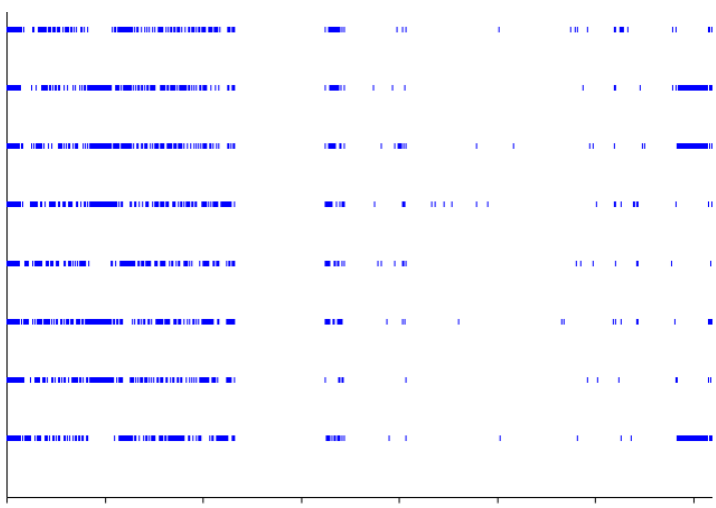

Figure 16: Bars correspond to a web service being called from different threads. Apparently there are periods when it takes longer, and periods when it's not called at all.

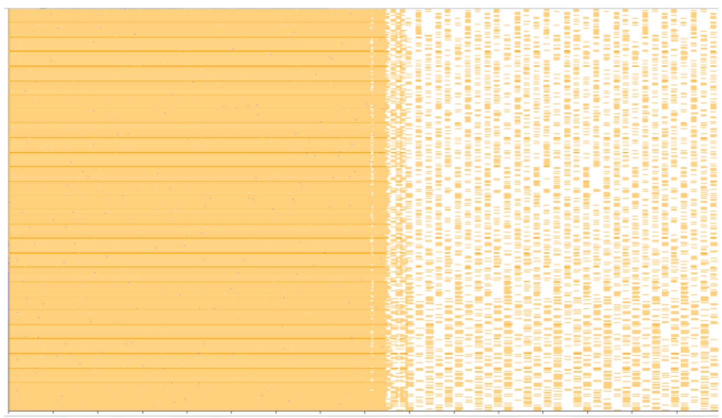

Figure 17: Several jobs run concurrently and saturate the cluster. Then all but one finish, and the one remaining runs in bursts of tasks, these bursts being not parallel enough to saturate the cluster (I recall it was a 480-core cluster and 160-task bursts).

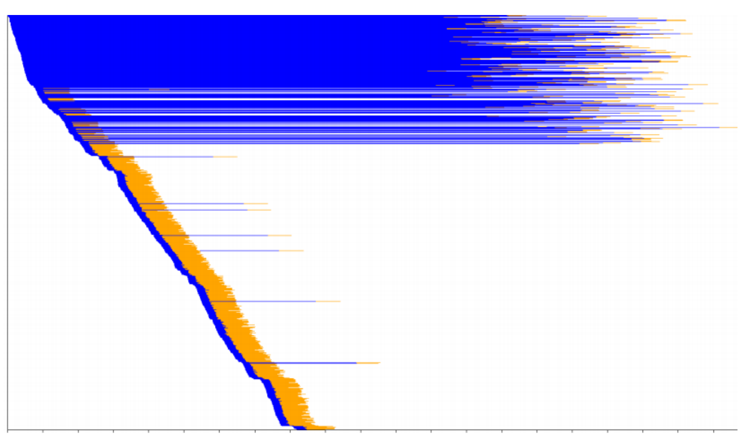

Figure 18: Orange: working, blue: fetching from memcached. We see that early calls to memcached take a ridiculous amount of time. Later calls sometimes take long too, but not that long (I recall the reason was a broken retry mechanism).

4 timeplot — drawing quantitative plots from event streams

While **splot** is designed to show you the *qualitative* patterns in a program's behavior, **timeplot** is meant to show *quantitative* patterns. It takes as input a sequence of events in a format similar to that of **splot** (but richer) and allows you to visualize the quantitative characteristics of that sequence in many different ways — e.g. to compare the frequency of different events per time unit, or to look at the distribution of event durations, etc.

The input to **timeplot** is a sequence of events of the form "event X happened", "durable event X started/finished", "a numeric or discrete variable P took value X". Every event has an *input track*.

The output is several plots vertically stacked on top of each other, with a common time axis. Every plot corresponds to one *output track*. Usually *input tracks* correspond to output tracks 1-to-1, though sometimes the relation is more complex (e.g. when we wish to compare values from several input tracks on one plot, or we wish a single input track to participate in multiple plots). The mapping is described in section 4.2. Every output plot is from one of the several available types (e.g. simple dot or line plot, or quantile plot, etc.). The kind of plot to draw on a particular output track is specified by the chart kind mapping, also described in section 4.2.

4.1 Motivation

4.2 Concepts

Event The atomic unit of information in the input trace. It tells can be one of several types: "something has happened" (this is called an *impulse event*), "something has started/finished" (this is called an *edge event*, and the activity delimited by start/finish is called a *long event*), "some parameter had a particular value" etc. The types of events correspond to what is usually found in typical program log entries. Every event happens on a particular *input track*.

Input track A named group of events in the input trace. Usually corresponds to a single parameter being measured, e.g. there could be an input track for request execution times named "rtime". Then we would have the input trace consist of events with track "rtime" and numeric type (see different types of events described in section 4.3).

Output track A named group of events in the output plots. The output track of an event is often equal to its input track, but in the general case is determined from its input track by the process of *track mapping*. All events with the same output track are shown on one output plot.

Output plot A single plot in the resulting picture. The picture consists of several output plots vertically stacked together with a common time axis. A single output plot is based on values from a single output track.

Track mapping The process by which events from different input tracks are mapped onto output tracks, e.g. to make events from several input tracks participate in a single output plot, or to make events from a single input track participate in several different output plots. It is described in section 4.2.1.

Plot kind The type of an output plot: e.g. dot plot, line plot, quantile plot etc. There are also a couple of "meta" plot kinds: duration plots and 'within'-plots. Plot kinds usually have parameters, e.g. the percentiles of interest on a quantile plot. Plot kinds are described in section **??**.

Plot kind mapping The process by which we determine what plot kind to use for visualizing a particular output track. It is based upon matching regexes against the input tracks of events mapped to this output track. The process is described in detail in section 4.2.1.

The following concepts are important for understanding the different event types and some plots produced from them:

Counter A logical time-varying variable associated with an *input track* which can be bumped by start/finish (*edge*) events. E.g. if your input trace includes events like "started/finished executing a request", then there's a logical counter that can be used to plot the number of concurrently executing requests.

Impulse event An input event without parameters that just denotes that something has happened. E.g. if you're interested in the number of completed requests per second, you can have an input trace with impulse events on the track "completed-requests" and draw an "event count" plot of that.

Edge event (counter bump) An input event without parameters that denotes that some activity (long event) has started or finished. It can at the same time be perceived as a bump of +1 or -1 to the logical counter associated with this event's input track.

Long event The logical activity delimited by a start and finish event. More precisely, the period during which a *counter* is greater than zero. The duration of long events can be measured (producing a bunch of numeric events) and you can draw all kinds of plots about these numeric events, e.g. if your input trace has "request started/request finished" events but doesn't have numeric events about request durations, you can still draw a quantile plot of request durations.

4.2.1 Track mapping

4.2.2

Counter Impulse and interval events Discrete and continuous variable Duration plots

4.3 Input format

4.4 Simple example

4.5 Chart kinds

4.6 Advanced features

4.7 Option reference

4.8 Gallery

5 Acknowledgements

The following people contributed code to the tools or provided help otherwise at different times.

- Tim Docker wrote the excellent **Chart** library, on which **timeplot** is based.
- Ivan Tarasov promoted the tools, hosted an event where I could talk about them, and made a fork with a quantile graph with a log scale.
- Arnaud Bailly found a large number of important bugs.
- Dmitry Astapov and Julia Astakhova gave a lot of feedback.
- Orion Jankowski did a pullrequest which exposed a number of bugs.
- Jason Dusek, Ilya Teterin, Julia Chertkova gave examples of their usage of **timeplot**.

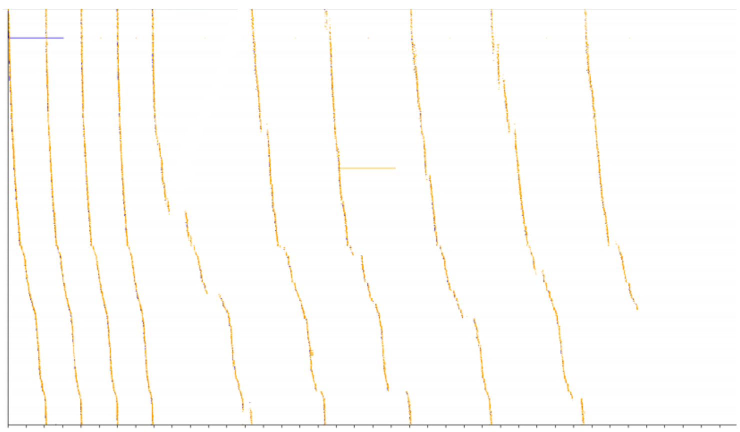

Figure 19: There are 1900 tracks (cluster cores) here. The tasks that we put into the shared queue are by far too short, so the queue (and perhaps the task producer) becomes the bottleneck. We also see that the queue feeds tasks to workers in round-robin. And we also see that it slows down over time. And small pauses in the queue or task producer cost an awful amount of computing time because everyone is waiting for them.