Santiago Pagola (sanpa993)
Hao-Hsiang Liao (haoli436)

TDDC78 Lab 5

Tools

Table of Contents

Introduction

The goal of this assignment is to try two different tools: a parallel debugger, called TotalView, and a tracing tool from Intel called Intel Trace Analyzer and Collector (ITAC hereafter).

Debugging with TotalView

The objective in this first task is to identify a bug with the help of the parallel debugger TotalView. This is a fully featured, feature-rich parallel debugger which allows users to debug parallel programs. It provides a nice graphical user interface, and many tools to debug existing errors in the code.

For this first task, we will simulate a very basic bug: dereferencing an invalid pointer! We will apply a patch designed for it, called 0001-Buggiest-bug-ever.patch, which can be applied to the whole project. We will place the bug on the first lab, i.e., one of the image processing filters: the thresholding filter.

With this said, once being remotely logged in into Triolith, we request a whole node for the simulation, and once we are allocated the desired resources, we run our program with the following command:

```
$ mpprun --totalview ./bin/thresc ../imgs/im1.ppm out1.ppm
```

When a pop-up comes up and asks if we want to stop this job, we say ‘Yes’ since we don’t want to let the program run, we want to debug it.

Next, we set a breakpoint where the bug is. In fact, when debugging a “real” bug, we don’t usually know where the bug is, so we set breakpoints at different points until the fault is found, but in this case since we are simulating one, we know where we should be looking. Breakpoints can be set by

selecting ‘Action Point’ in the toolbar and clicking on ‘Breakpoint’. Here is a snapshot of our TotalView at this step:

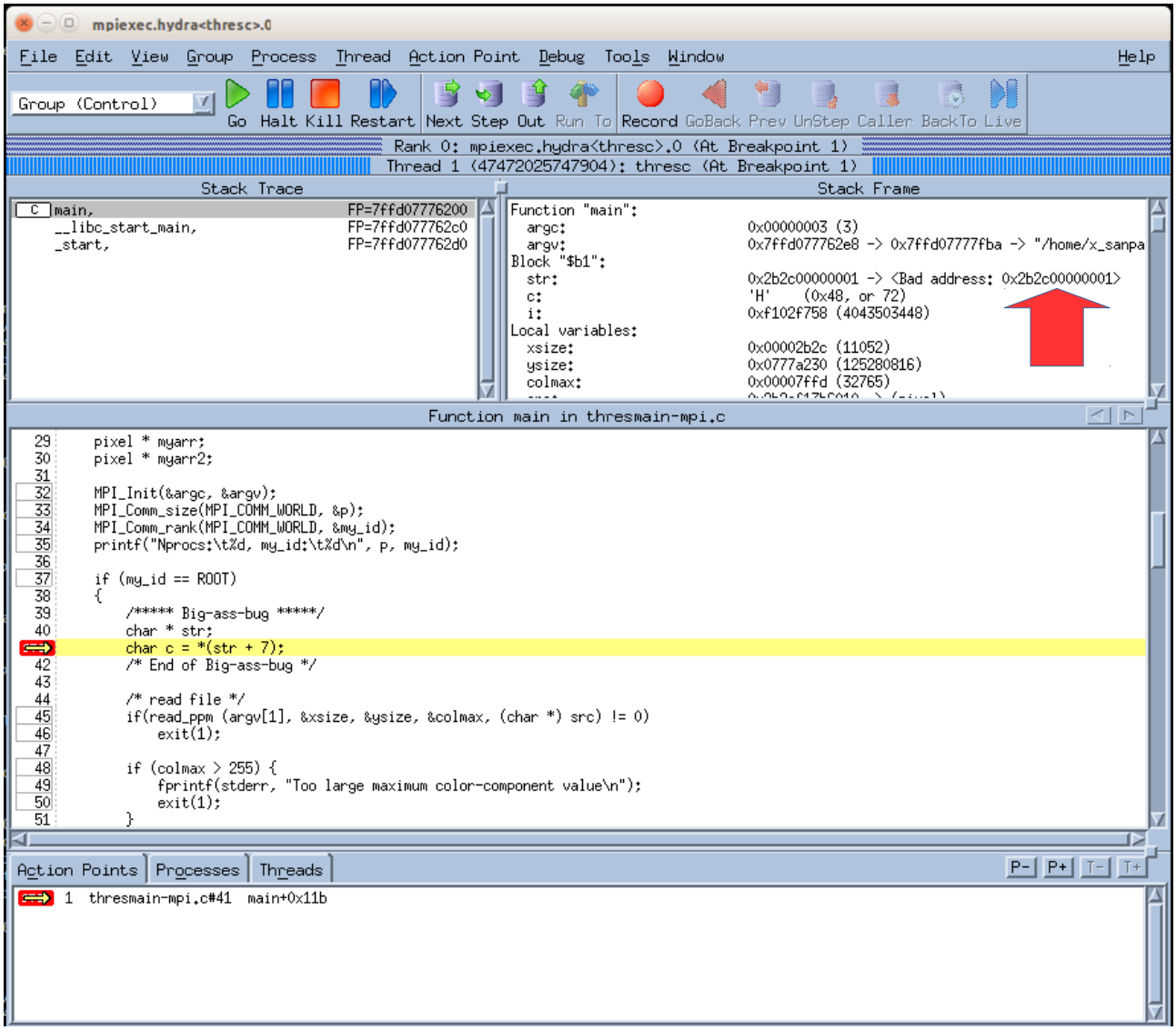

Illustration 1: Screenshot of TotalView

We can see from the Stack Frame that the variable `str` has an invalid address (pointed by the red arrow), what we suspected. This way, TotalView shows us all the variables in the current frame for the current process, and by looking at them it’s easy to spot the bug.

We added some more breakpoints at different lines of the code, between which we can jump by clicking on ‘Next’, or we can even step into the current statement by clicking ‘Step’. For this simple bug, we didn’t have to step into any function.

Santiago Pagola (sanpa993)
Hao-Hsiang Liao (haoli436)

Tracing MPI programs with ITAC

The second part of this lab is about tracing an MPI program with ITAC. The lab of choice has been the first one, again, but the blurry filter will instead be analyzed, since there it is computationally bigger than the thresholding filter. With the help of ITAC's graphical interface, we will be able to perform a bottleneck analysis in order to determine which parts of our code are more time-costly.

Let's start by analyzing our defined traces in the code:

```
int main_class;
/* Define our VT class */
VT_classdef("Blurry filter", &main_class);
/* Define handles for different states */
int hf, vf, h2v, comm;
VT_funcdef("Horizontal Filtering state"  , main_class, &hf);
VT_funcdef("Vertical Filtering state"    , main_class, &vf);
VT_funcdef("Horizontal to Vertical state", main_class, &h2v);
VT_funcdef("Communication state"         , main_class, &comm);
```

The very first thing we do is to define a class using Intel's API, called Blurry Filter. After the call to `VT_classdef()`, we obtain a handle it. Next, we defined custom execution points (or "functions" in Intel's terminology) within our created class which will allow us to trace specific execution points in our code. In this case, we defined four different functions: the horizontal filtering part, the adjacent matrix translation, the vertical filtering and the overall MPI communication of the program. Note that these definitions must be done once the MPI environment has been initialized, i.e., after `MPI_Init()` has been called. We experienced some problems when trying to collect the traces with ITAC and the problem was due to declaring these classes before calling `MPI_Init()`. With this said, before and after every call to these functions of interest, calls to `VT_enter()` and `VT_leave()` are done in order to get traces in ITAC later on:

```
VT_enter(h2v, VT_NOSCL);
/* Convert our original image array `src` to a matrix */
from_array(src, xsize, ysize, &original);
/* Calculate its adjoint matrix */
adj_matrix(&original, &adjoint);
/* Convert the adjoint back to the array in `src_adj` */
to_array(&adjoint, src_adj);
VT_leave(VT_NOSCL);
```

After compiling the program with the necessary compile and link flags, we run the simulation of the blurry filter with 8 cores, the third provided image and a radius of 5 with the following command:

```
salloc -n8 mpprun --pass="-trace" ./bin/blurc 5 ../imgs/im3.ppm out.ppm
```

When the simulation is done, we open the ITAC main user interface with the generated trace file, and this is what the load balance with our defined class looks like:

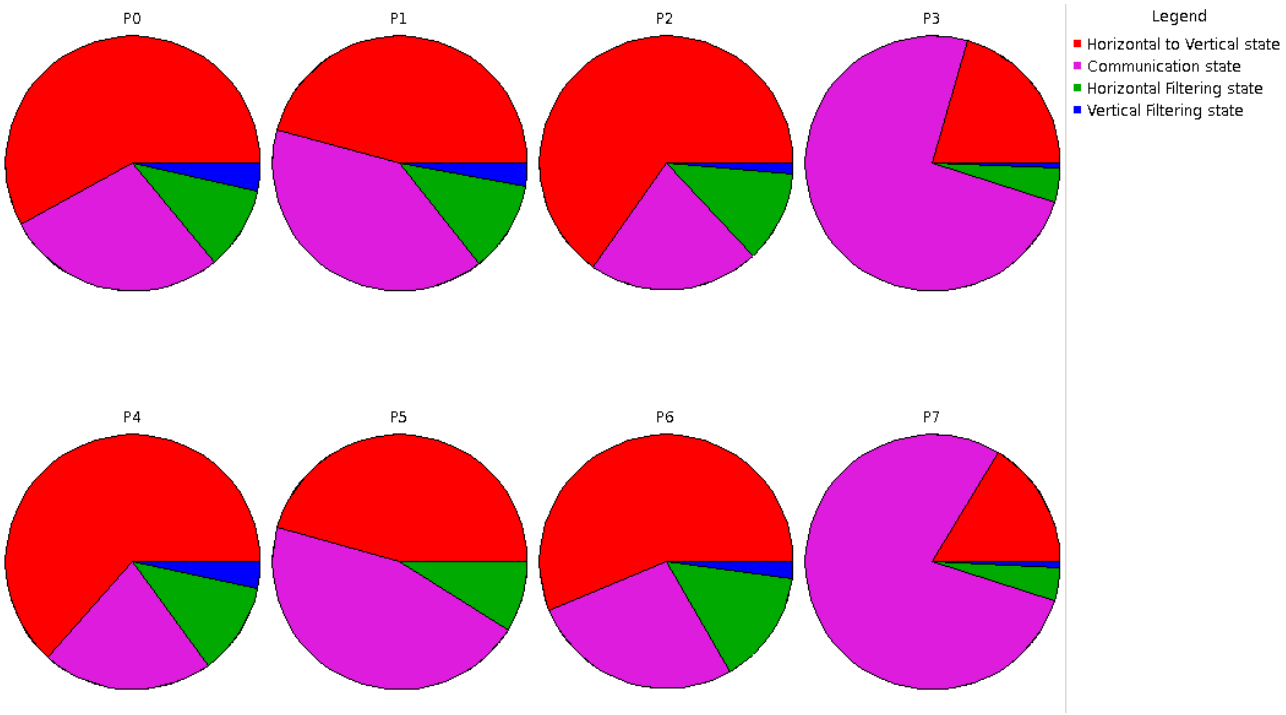

Illustration 2: Load balance of our own-defined trace points

As can be seen from Illustration 2, the program was run with 8 cores and the timing splits between the four defined functions is shown. Note that the communication state (in magenta) takes most of the time in P7, P5 and P3, but for the rest of the processes the actual adjoint matrix calculation is what is most time-consuming. This may be due to e.g. these processes staying in an `MPI_Barrier()` call longer than the rest, i.e., they came to the barrier first and had to wait for the other processes in the group to reach the barrier. Another reason, which will be discussed shortly, is that these processes reached a call to `MPI_Gather()` before the `ROOT` process, i.e., where the results are to be "gathered".

Another useful view of ITAC is the event timeline, which looks like this:

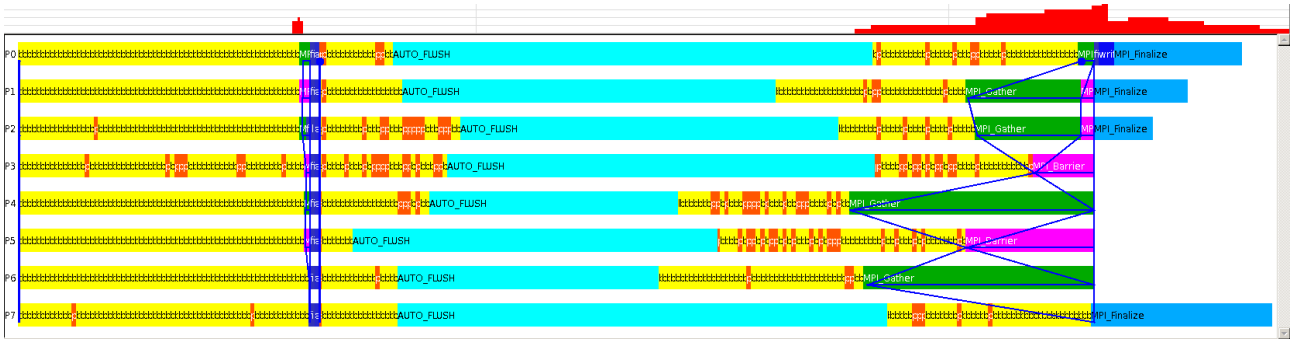

Illustration 3: Full event timeline, with all function groups shown.

The legend is not shown in the figure above, but this chart represents all traced functions in the execution of our program. This chart can be filtered by function groups, by choosing 'Advanced'→ 'Function Aggregation' and selecting a specific group.

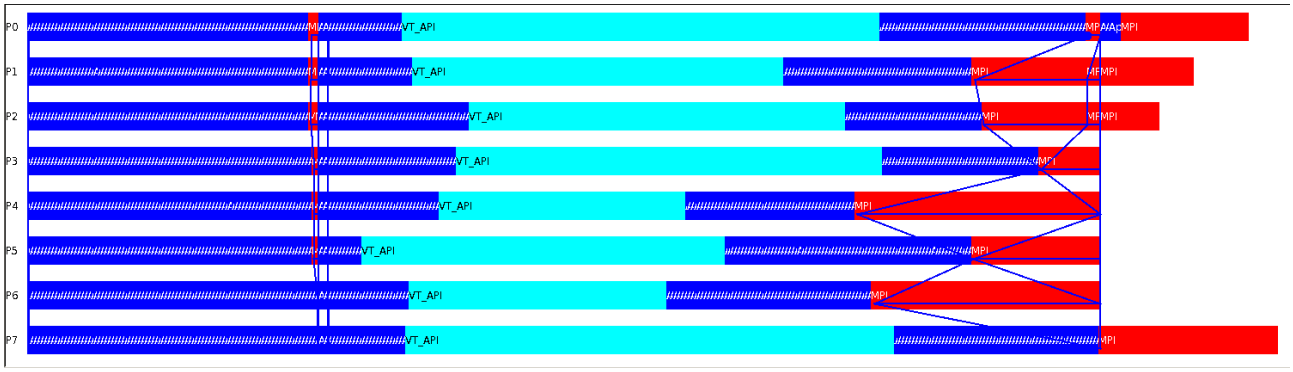

Illustration 4: Group event timeline, with all major function groups: Application (blue), VT_API (cyan) and MPI (red).

Illustration 4 is a filtered version of Illustration 3, where instead of showing every single function of the program in a timeline, function groups are shown. If we further filter the above chart and focus only on the MPI functions, the graph below is obtained:

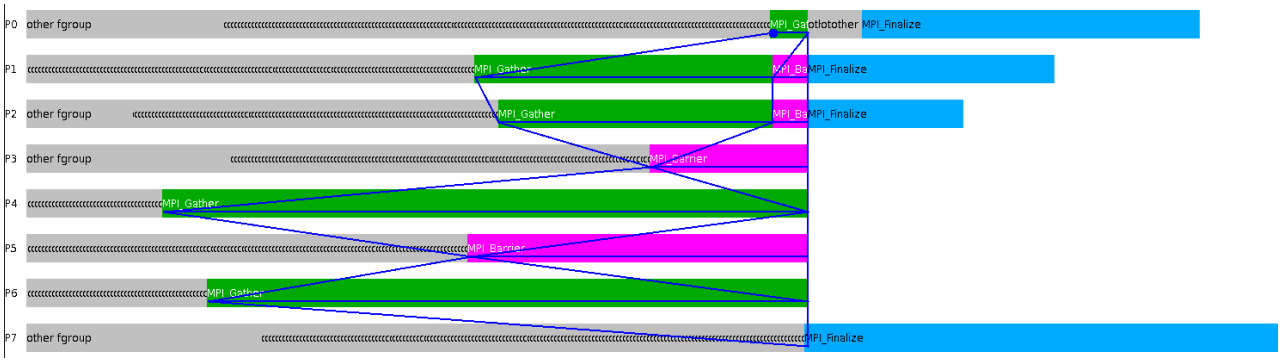

Illustration 5: MPI event timeline, only shown the last part of the program execution.

Note that this event timeline isn't shown from the beginning, we are only focusing on the last part, where the last call to `MPI_Gather()` occurs. We can indeed confirm our suspicions with processes P3, P5 and P7, i.e., how they spent a significantly bigger time in communication with respect to the other processes: P7 took the largest time to end the call to `MPI_Finalize()`, P5 stayed the longest time on the `MPI_Barrier()` call (it was the first process to enter the synchronization point), and P3 stayed the second longest time in the barrier.

Lastly, there are two very interesting charts in ITAC, namely the *qualitative* and *quantitative* timelines. These offer another way to represent the parallel behavior of the application. The *quantitative* charts show how many processes of threads are involved in which function, whereas the *qualitative* charts show event attributes such as data volume of messages as they occur over time.

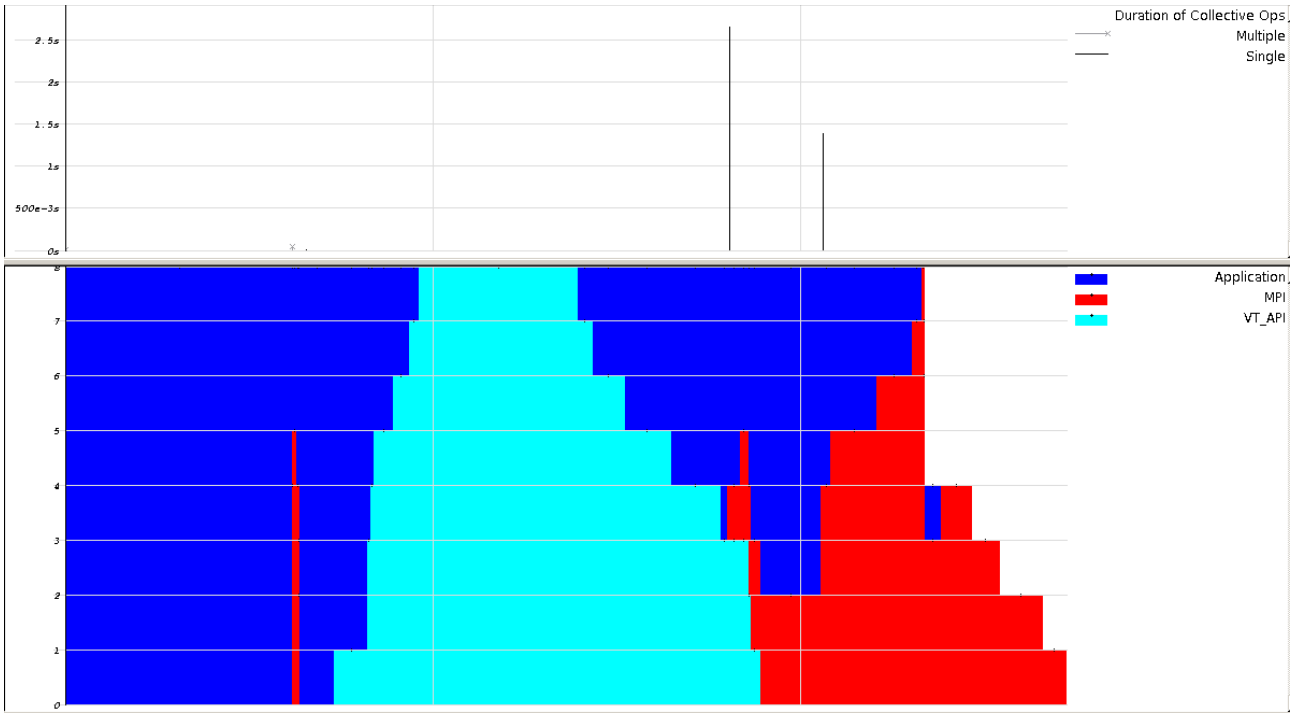

Illustration 6: Qualitative (top) and quantitative (bottom) charts of the major function groups in the application.

From bottom part of Illustration 6, i.e., the *quantitative* timeline, we can see how much time each process spends on each function group, namely Application, MPI or VT_API. As with the timeline snapshots in Illustrations 3, 4 and 5, the contents of can be further filtered and show only a specific function group. Here is what happens when we filter the contents of Illustration 6 to only show MPI related functions:

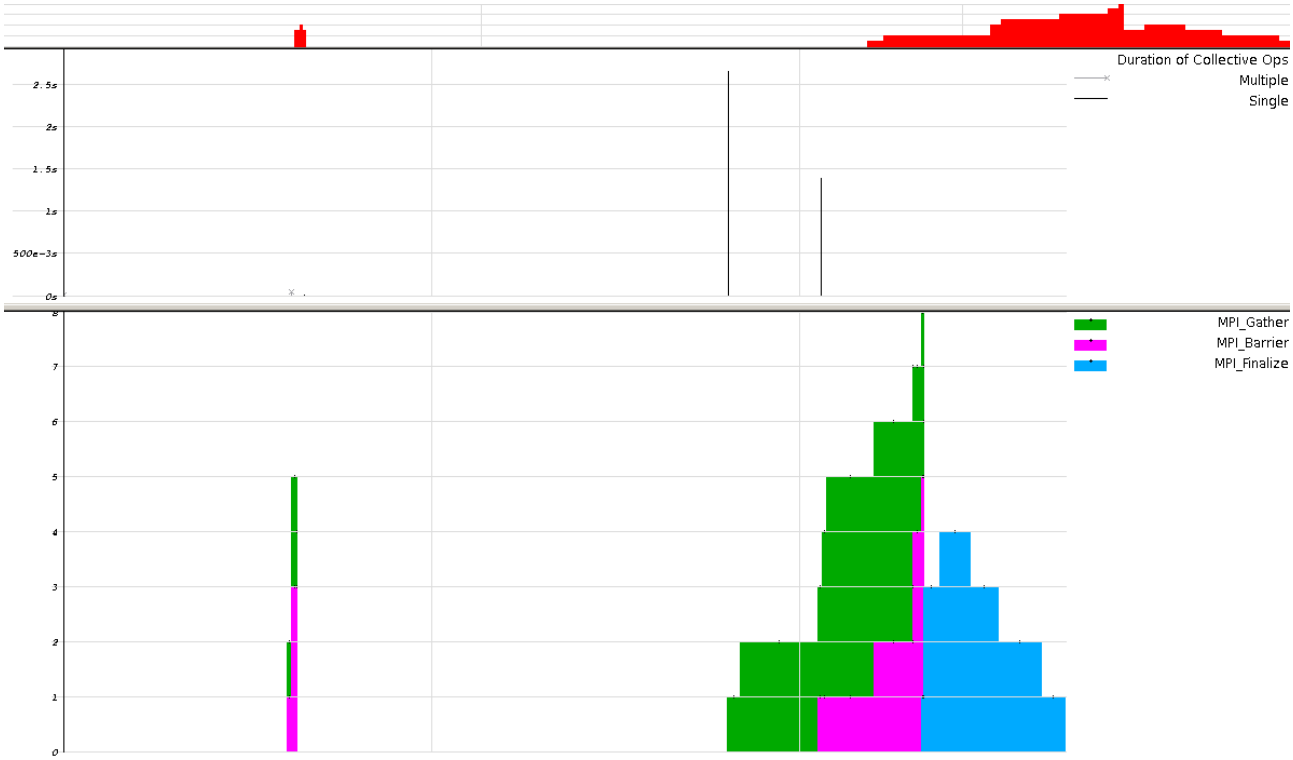

Illustration 7: Qualitative (top) and quantitative (bottom) event timelines for the MPI function group.

It becomes much clearer to see where the `MPI_Barrier()` is (the magenta-colored regions) and which processes came to the barrier first and stayed longer.