Guided demo MPI analysis:

- Load trace `mpi/HydroC_mpi64.prv.gz`
- Load configuration file `cfgs/mpi/mpi_stats.cfg`
    - This configuration pops up a table with %time of every thread spends in every MPI call.
    - Look at the global statistics at the bottom of the outside mpi column. Entry *Average* represents the application parallel efficiency, entry *Avg/Max* represents the global load balance, entry *Maximum* represents the communication efficiency.
    - Change in the main window the metric to *nb of calls* to look at the number of invocations and *Average duration* to look at the average duration of each MPI call
    - Open the Control Window from the table (top left button). Use the zoom and contextual menu (right click) to navigate, view communications lines...
- Load configuration file `cfgs/mpi/user_functions.cfg`.
    - This configuration pops up a timeline where every color corresponds to one of the instrumented user functions,
    - Zoom to identify one iteration and synchronize the interval with the MPI call timeline using the *Copy* and *Paste* commands.
    - Double click on one color over the timeline to open the textual display of the actual function name and duration. Use the menu option *Info Panel* to hide it again.
    - Create a table showing the default statistic (Time) accumulated per user function.
- Load configuration file `cfgs/mpi/2dh_usefulduration.cfg`
    - This configuration shows a histogram of the duration for the computation regions. The computation regions are delimited by the exit from an MPI call and the entry to the next call.
    - Using the *Open Filtered Control Window* button select one of the histogram modes, this will create a new timeline with the selected duration range. Use the *Fit Semantic Scale →Fit Both* option to set the gradient color such that it fits the actual range of durations in the new window.
    - Modify the Statistics of the useful duration histogram to use the *correlate with metric* and verify that the metric wiindow is *Instructions per cycle*. Now the cell

color will correspond to the IPC showing the correlation between duration (position) and IPC (color). Zoom into an unbalanced mode to verify if the unbalance is related to a different IPC.

- Load configuration file `cfgs/mpi/2dh_useful_instructions.cfg`
    - This configuration pops up a histogram of the instructions on the computing regions (outside MPI).
    - Copy the scale of one iteration from the useful duration timeline to this new histogram. Vertical lines identify well balanced regions with respect to the number of instructions.

Futher questions:

Paraver profile analyses

- Load configuration file `cfgs/mpi/2dp_uf_several_stats_numbers.cfg` Study the three different profiles. What do they compute (control window, statistic, data window)?
- Are all routines equally balanced in terms of time?
- Are all routines equally balanced in terms of instructions? Any of them is worse balanced?
- Do all routines achieve the same IPC. Which seem to have bad IPC?
- Check if L2 miss ratio is a possible cause of the observed poor IPCs (`cfgs/mpi/L2D_miss_ratio.cfg` displays the number of L2D misses per 1kinstr). Hint: clone the 2DP-uf-IPC profile and change data window.
- Routine equation of state:
    - Average duration per call? Is the granularity enough to parallelize with OpenMP, GPU?
    - How many invocations?
- Routine riemann:
    - Average duration?
    - Average IPC?
- Which routines call MPI and which not? Use `cfgs/mpi/2dp_uf_percentMPI.cfg`.
- Scaling: Load also trace of 128 processes (`mpi/HydroC_mpi128.prv.gz`) and `cfgs/mpi/2dp_uf_several_stats_numbers.cfg`

    - Pop up and synchronize the user functions views. Does it scale well?
    - Which routines scale well or bad? ($\eta=T_{64}/2*T_{128}$)
    - Do they scale well in terms of number of instructions?

Paraver histogram analyses

- Load `cfgs/mpi/3dh_several_uf.cfg`
- Routine UpdateConservativeVars
    - Imbalanced? Multimodal distribution?
    - Which instances take more time and which less?
    - Is it due to differences in the number of instructions?
- Routine equation_of_state
    - Are all the invocations of the same duration?
    - Is there a pattern?
    - Reason for difference between them?
- Routine riemann:
    - All invocations of the same duration? Multimodal? Reason?
    - Is the distribution similar at 64 and 128 processes?
- Routine make boundary
    - What is its average IPC? What is the average IPC of the computation outside MPI? Why are they so different?

Guided demo Dimemas analysis:

- In directory Dimemas we have prepared:
    - The original `HydroC_mpi64.prv.gz` and its translation to the Dimemas format `HydroC_mpi64.trf`
    - Dimemas configuration files and the corresponding traces generated by the Dimemas simulation for:
        - `dimemas_files/64.nominal.cfg`: a “nominal” setup: 8 processes per node, one adapter per node, 1GB/s, no network contention. `D.64.nominal.prv.gz`
        - `dimemas_files/64.10xCPUr.cfg`: assuming 10x faster CPU for all computations. `D.64.10xCPUr.prv.gz`
        - `dimemas_files/64.10xCPUr.1xEOS.cfg`: assuming 10x faster CPU except for routine equation_of_service that is assumed to stay equal. `D.64.10xCPUr.1xEOS.prv.gz`
        - `dimemas_files/64.10xCPUr.0.1xEOS.cfg`: Assuming 10x faster CPU except for routine equation_of_service that is assumed to actually slow down by 10x due to parallelization overheads. `D.64.10xCPUr.0.1xEOS.prv.gz`
- Load the configuration file `cfgs/mpi/user_functions.cfg` on each of them. The loaded views will show the structure and relative impact of the different regions in the total time of each case. Copy the time scale from the original trace to each other trace to compare global scalability.

Further Dimemas analysis

- A similar process can be done using configuration files:
    - dimemas_files/64.ideal.cfg: An “ideal” setup with instantaneous communication. `D.64.ideal.prv.gz`
    - dimemas_files/64.NxCPUr.MBW.cfg: configuration of hypothetical target machine with N times faster CPUs and M (0.1 and 0.01) times the original bandwidth. `D.64.100xCPUr.prv.gz`, `D.64.100xCPUr.0.1xBW.prv.gz`, `D.64.100xCPUr.0.01xBW.prv.gz`

Guided demo MPI+CUDA analysis:

- Load trace `mpi+cuda/cuHydro_mpi32.filter1.prv.gz`
    - This tracefile contains only the MPI calls, user function and “large” computing burst from the MPI+CUDA execution. It has been obtained from the original trace with the filtering utilities in Paraver.
- Load configuration files `mpi/mpi_stats.cfg` and `mpi/user_functions.cfg`
    - Use them to compare the execution of the MPI and the MPI+CUDA version. Compare the structure within one of the iterations on the two versions.
- Load trace `mpi+cuda/cuHydro_mpi32.chop1.prv.gz`
    - This tracefile contains approximately a chop of one of the double iterations.. It has been obtained from the original trace with the cutting utilities in Paraver.
- Load configuration files `cfgs/mpi+cuda/3dp_cudakernel_uf.cfg` and `cfgs/mpi+cuda/cuda_events.cfg`
    - The first configuration shows a profile of the number of cuda kernel invocations by each user function. Use the *3D – Plane* chooser in the general Paraver window to select a different user function.
    - The second configuration shows the CUDA runtime events instrumented.
    - Use the zoom into a small area (control+select area) in one of them and synchronize all the timelines (*Copy* and *Paste Default Special*).

Further CUDA analysis

- Load the configurations `cfgs/mpi+cuda/2dp_cudakernel.cfg` and `mpi/2dp_uf.cfg`
- What is the average duration of the different kernels? And user functions?
- How many invocations of the different kernels appear? And user functions?
- Pop up the view cuda events and zoom to show just the two lines corresponding to process 1. Use *Paste Default Special* to see the exact same region and objects in the cuda kernel view.
- Look for the cuRiemann function invocations.

  - How many Kernel invocations per cuRiemann function are done? How long do they take? Is the overhead relevant? Is the GPU speed-up versus the sequential code good for this routine?
- Look for the cuEquationOfState function
  - How many Kernel invocations to the cuEquationOfState function are done? How long do they take? Is the same all over the trace? Is the overhead relevant?