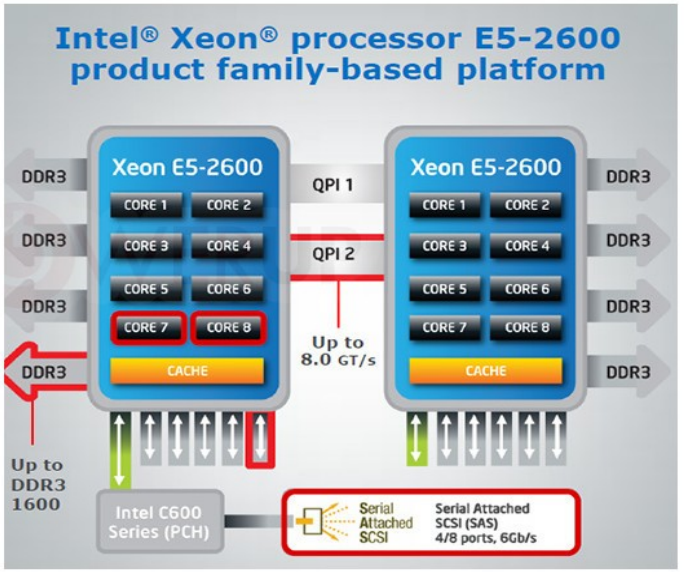
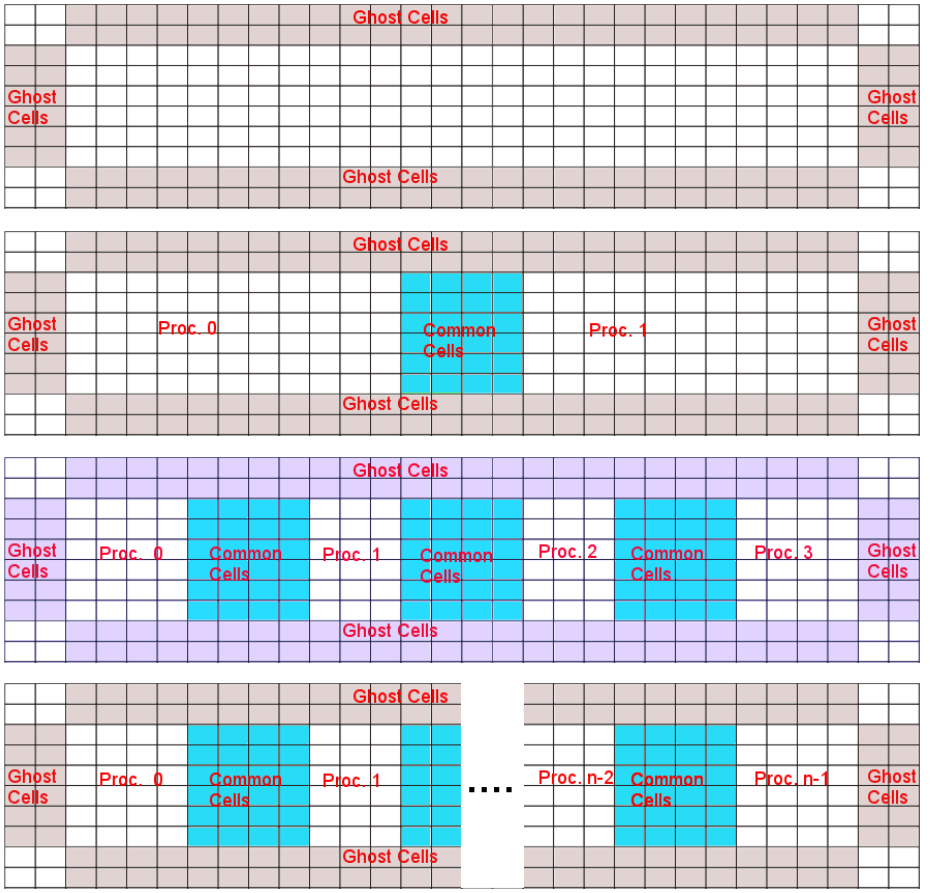
Introduction

Parallelization of a 2D Hydro code using MPI
Course: ESC401 - HPC1b: Parallel Computing
Authors: Rafael Küng, Christian Reinhardt and Mihai Tomozeiu

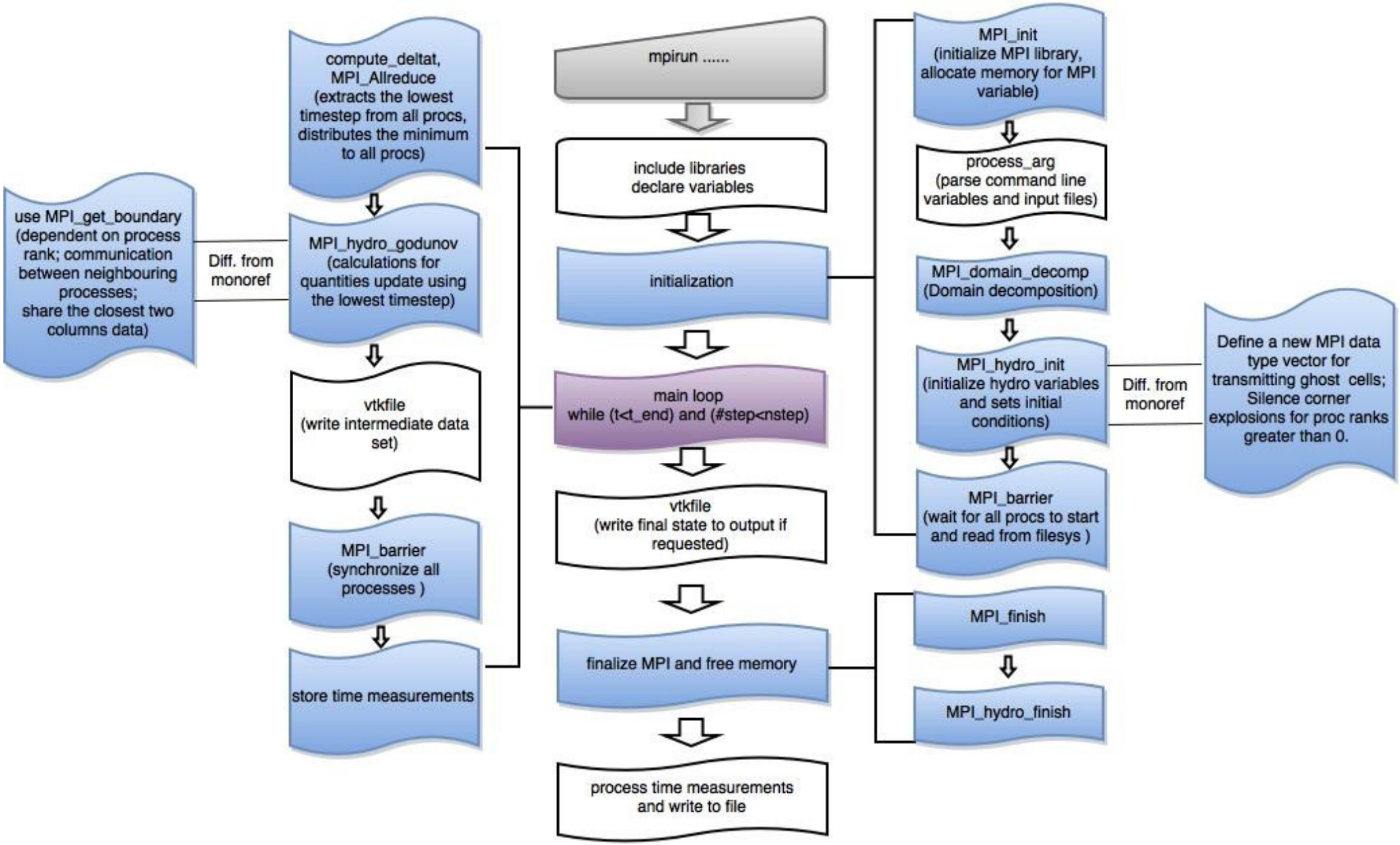
- The original code simulates an explosion in the lower left corner (right images) of a 2D wind tunnel using a single process
- Number of individual proper cells n_x(rows) * n_y(columns) and 2 * (n_x +n_y) ghost cells for reflective boundaries
- Approach:
  - general idea: modify as little code as possible
  - slicing the domain along the x-axis
- Domain distribution in right figures panels:
  - 2nd: 2 cores; 3rd: 4 cores; 4th: n cores
- Each neighboring processes need to communicate the closest two columns of proper cells to each other
- Minimum time step found among processes will become the time step of all processes



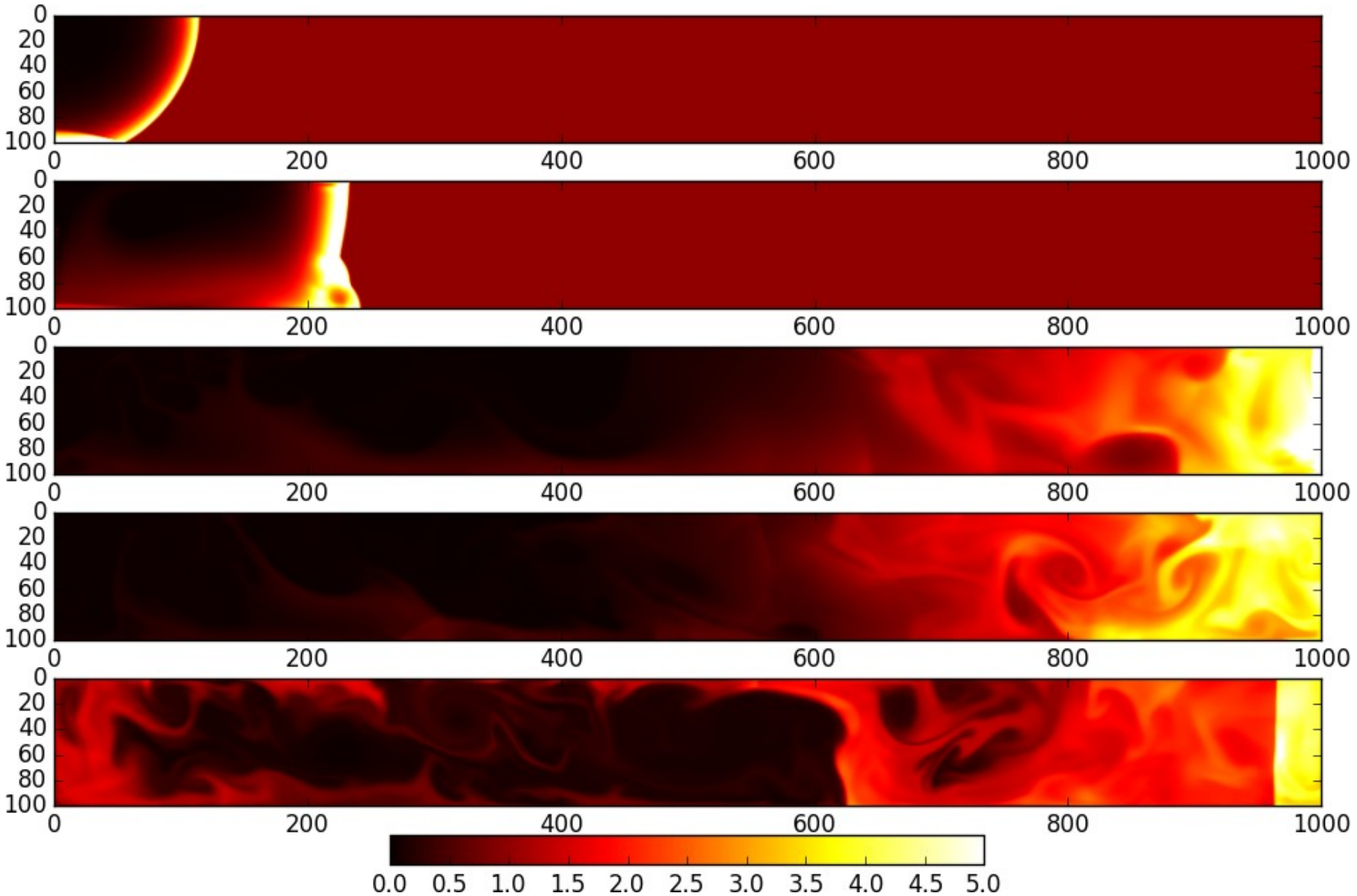


- The simulations were run on the local zbox4
- Intel Xeon E5-2660 (family platform sketch in the left image)
- 8 cores per socket
- 2 sockets per node
- Internode communication with QDR Infiniband

Code Description



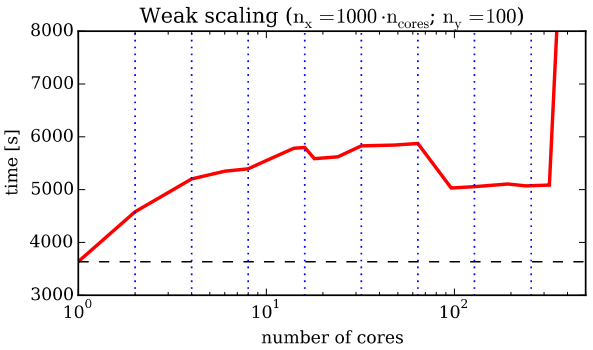
Results / Output

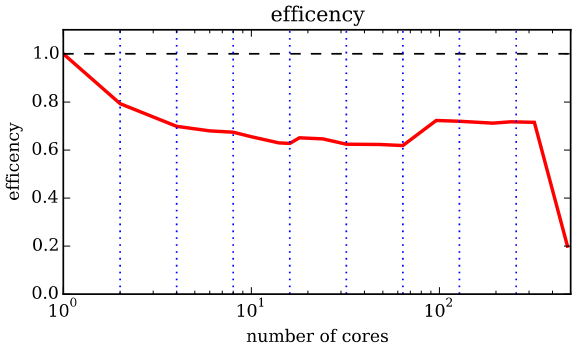


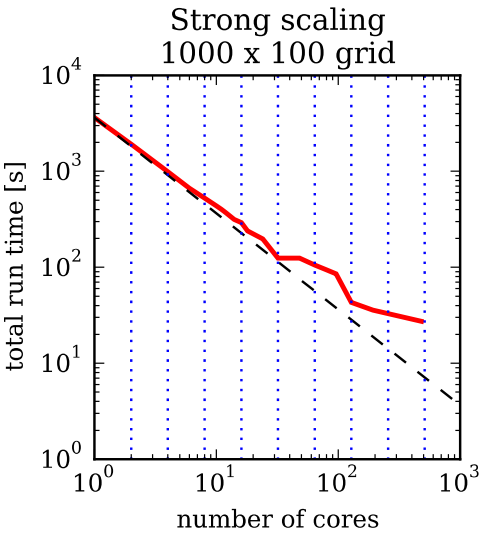
Example of a run (1000x100 grid): An initial overpressure in the top left cell causes a blast wave to propagate trough the computational domain. The walls are reflecting so the shock bounces back at the domain boundaries and interferes with the original wave. This causes an oscillating bulk motion with beautiful and quite complex wave patterns in the inner part of the domain. The colors represent the fluid density where black is zero and white is at least five.

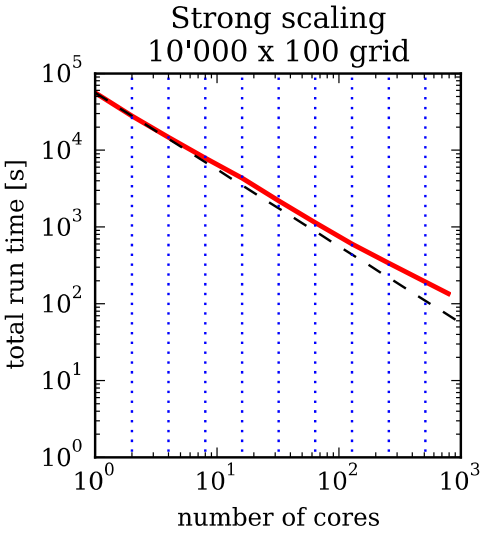
Scaling Behaviour

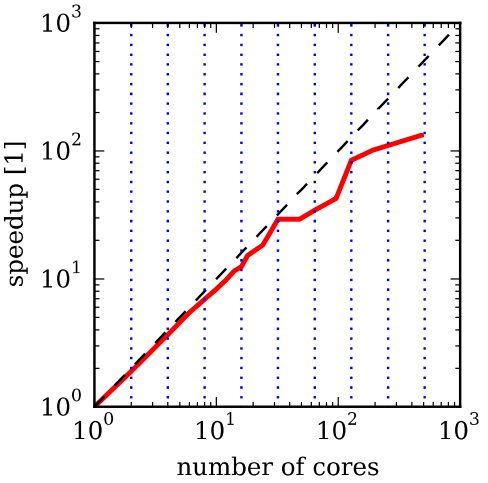
- Strong Scaling (right figures)
  - At large numbers of cores (approaching 500) the great majority of the cells have to be shared among neighbouring processes.
  - Above 500 cores the code breaks down since the number of columns per process becomes less than 2. A minimum of 2 columns are needed for proper data transfer between neighbouring processes.
  - Please note: for the 10'000 x 100 grid: The value for 1 core is interpolated
- Weak Scaling (left figures)
  - n_x scaled with the number of cores, n_y constant
  - the 16 cores point marks the transition to internode communication showing a core performance worsening
- Legend: blue dotted lines show powers of 2 black dashed line shows ideal scaling behaviour

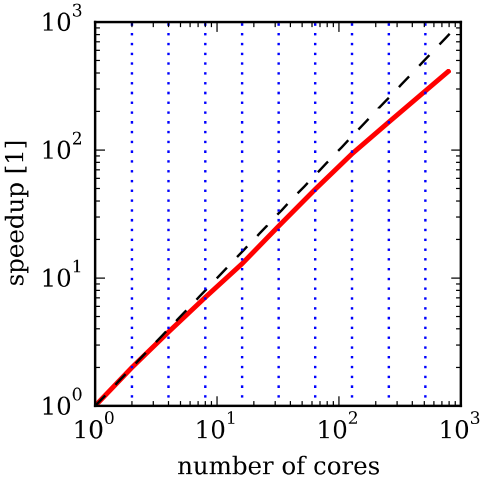


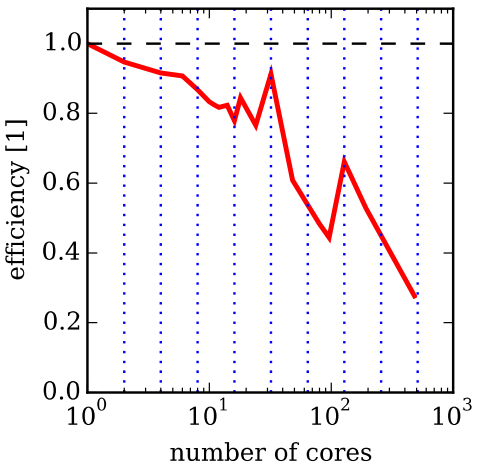


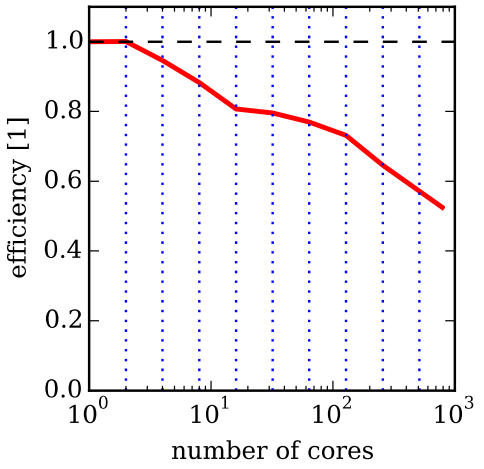




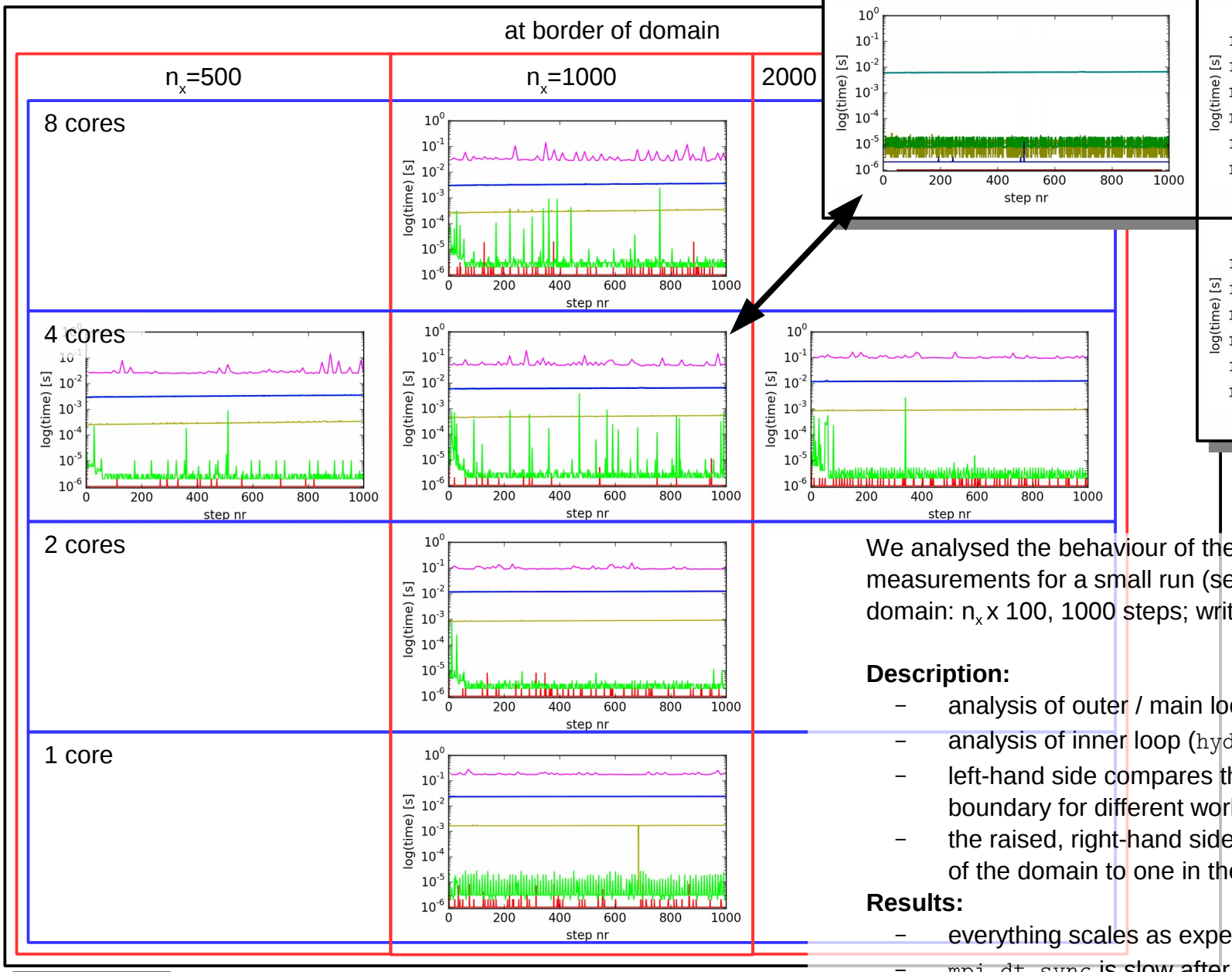


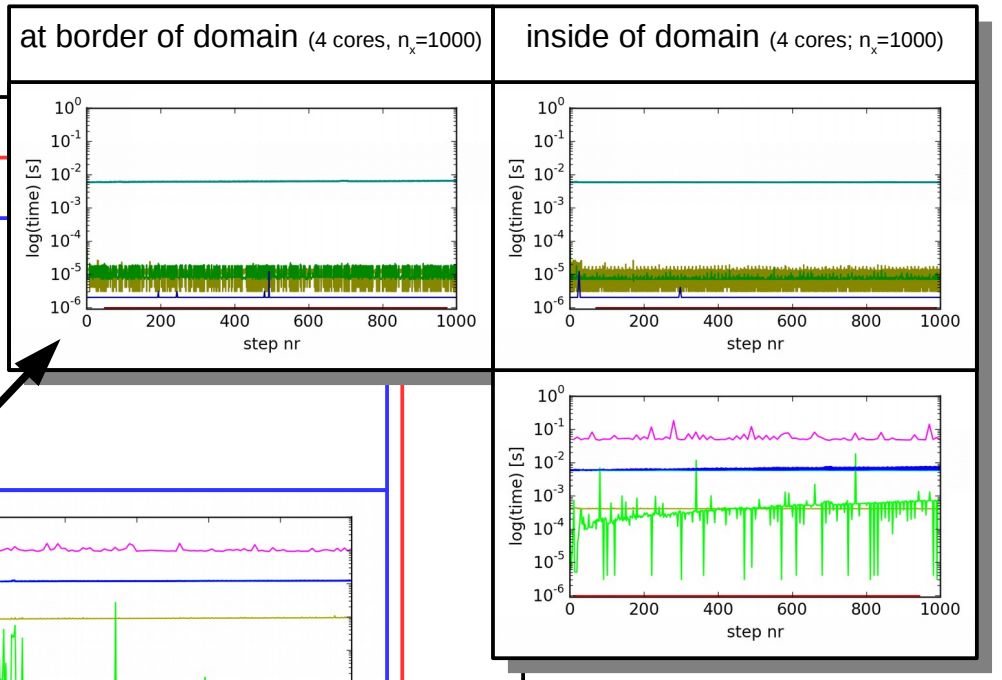






Code Diagnostics – Overview





We analysed the behaviour of the code using several timing measurements for a small run (settings used: run on zbox4; domain: n_x x 100, 1000 steps; write output each 10th step)

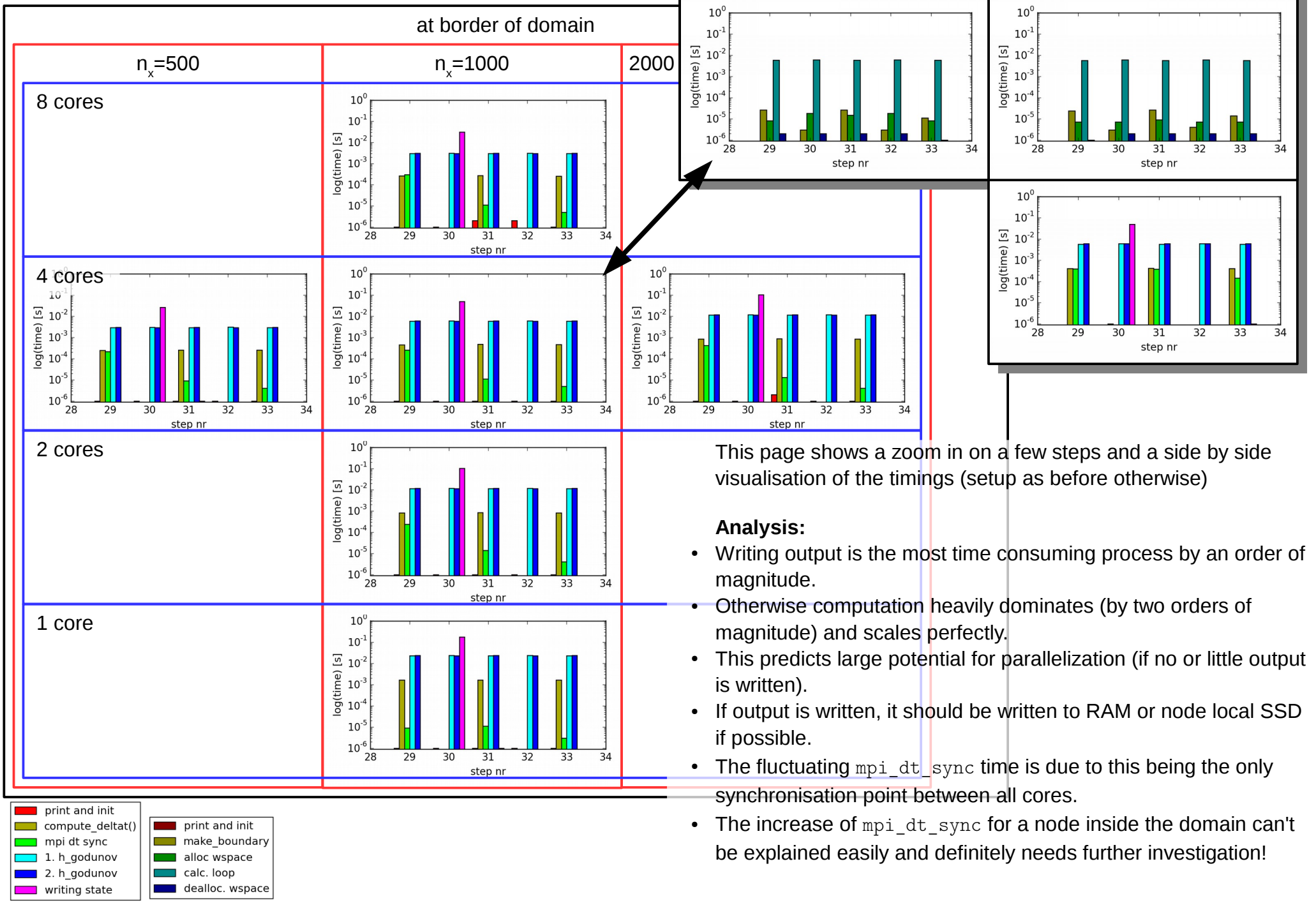
Description:

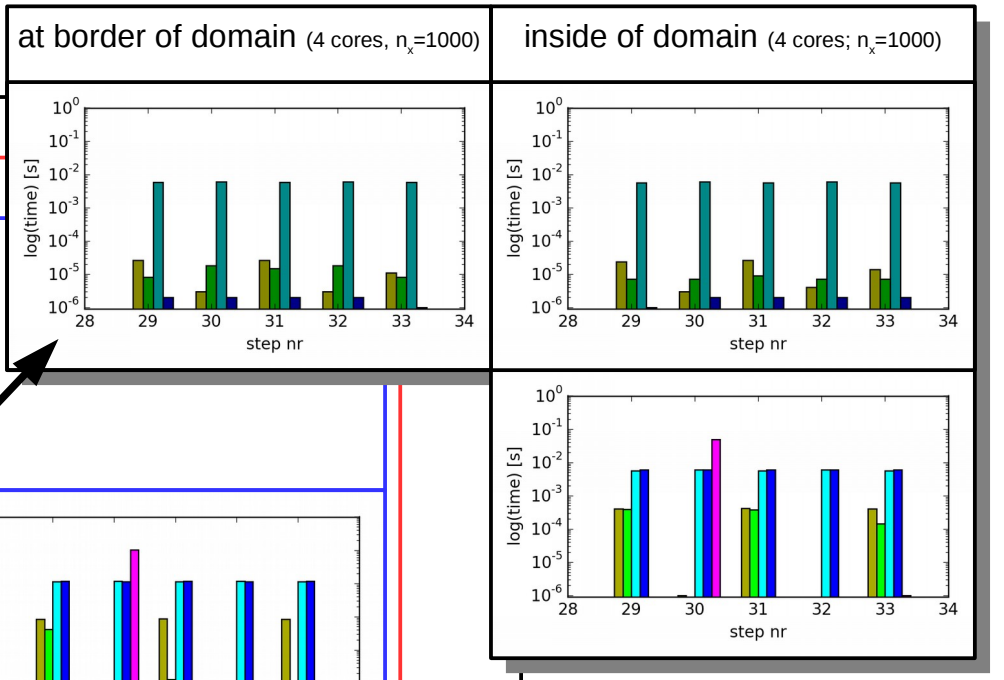
- analysis of outer / main loop with bright colors
- analysis of inner loop (`hydro_godunov()`) in dark colors
- left-hand side compares the behaviour of a core at the boundary for different world sizes and domain sizes.
- the raised, right-hand side compares a core at the border of the domain to one in the middle.

Results:

- everything scales as expected
- `mpi_dt_sync` is slow after startup, noisy and increases for a node inside the domain
- MPI code inside `make_boundary()` doesn't show this behaviour.

Code Diagnostics – Details





This page shows a zoom in on a few steps and a side by side visualisation of the timings (setup as before otherwise)

Analysis:

- Writing output is the most time consuming process by an order of magnitude.
- Otherwise computation heavily dominates (by two orders of magnitude) and scales perfectly.
- This predicts large potential for parallelization (if no or little output is written).
- If output is written, it should be written to RAM or node local SSD if possible.
- The fluctuating `mpi_dt_sync` time is due to this being the only synchronisation point between all cores.
- The increase of `mpi_dt_sync` for a node inside the domain can't be explained easily and definitely needs further investigation!

Conclusions

- We successfully parallelized the existing serial code with MPI.
- Using our code we ran several simulations with different grid sizes on a varying number of CPU cores. Both the hard and the weak scaling plots show a clear performance improvement when run of multiple cores.
- The analysis shows that hard scaling up to 480 cores works (but deviates slightly from ideal)
- Outputs are - as expected - a bottleneck and reducing outputs does indeed increase performance.
- In general we are limited in the maximum number of cores to $n_x/2$ because we need at least 2 columns in the inner part of the computational domain.
- We tried to add OpenMP, but didn't invest much time:
  - We failed to parallelize the outer loop, probably because of faulty declaration of variables.
  - Parallelizing the inner loops succeeded, but performance was inferior, it almost halfed.
  - This is to be expected, the overhead of spawning a thread for each small loop inside the main loop is too much.