High Performance Computing

Parallelization of a 2D hydro code using MPI and OpenMP

Yannick Boetzel
University of Zurich

Hydro code

The hydro code solves the Euler equations on a discretized grid of size nx * ny. The boundary conditions are given by 2 ghost cells on each boundary, so the total gridsize is (nx + 4) * (ny + 4). The code can be split into two main parts, the initialization and the main loop.

Initialization:

- Variables are declared.
- Grid is initialized with a point explosion in the lower left corner.
- Workspace is allocated.

Main loop:

- Time step is computed every even step.
- Grid is updated using a godunov-solver. This is a 1D solver for the Euler equations which is called first for each row to update the fluxes going into cells left and right, then for each column to update fluxes going into cells up and down. To decrease artifacts, the grid is updated two times for each computed timestep, once with the godunov function going rows-columns, then columns-rows.
- If requested, the current state is written to a vtk-file every n-th step.

Parallelization

We use MPI to parallelize the code. The grid is split into N segments, each computed by one core/process. There were four main aspect to be considered.

Initialization:

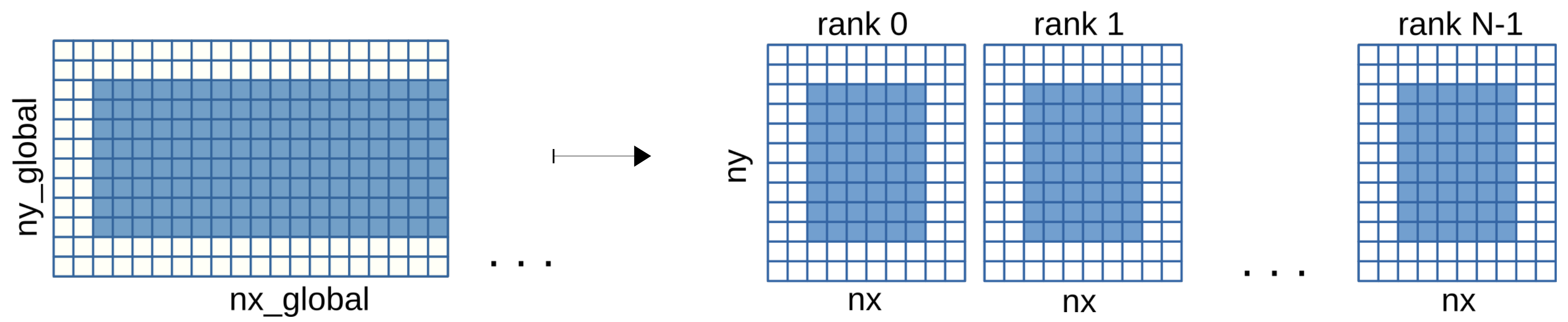
- The grid is split horizontally. Each process gets a segment of size

  nx = nx_global / world_size + (int)(rank < nx_global % world_size) ;   ny = ny_global

  This assures that the workload is distributed evenly. E.g. if the global grid has a width of nx_global = 100 and we have three MPI processes, it gets split up into segments of width 34, 33 and 33.

- Each process then is initialized with a grid of size (nx + 4) * (ny + 4), accounting for ghost cells. The ranks go from 0 to N-1 from left to right.

- The process of rank 0 is initialized with a point explosion in the lower left corner, all other ranks have a uniform grid.



Timestep:

- Each process computes its own timestep. To make sure each part of the grid is evolving at the same speed, we need to synchronize this timestep across all processes. We do this by a simple call of the function

  ```
  MPI_Allreduce(&dt, &dt, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD)
  ```

  This function automatically gathers all timesteps, finds the minimal value, and broadcasts it again to all processes.

Output:

- Each process knows only about its part of the grid. To output to a vtk-file, all segments need to be gathered, stitched together and written to a file. As output is rare, we chose a simple implementation: All ranks > 0 send their part of the grid to rank 0 via MPI_Send(…), while rank 0 receives all data via MPI_Recv(…) and writes it to a temporary buffer. The buffer is then written to a vtk-file.

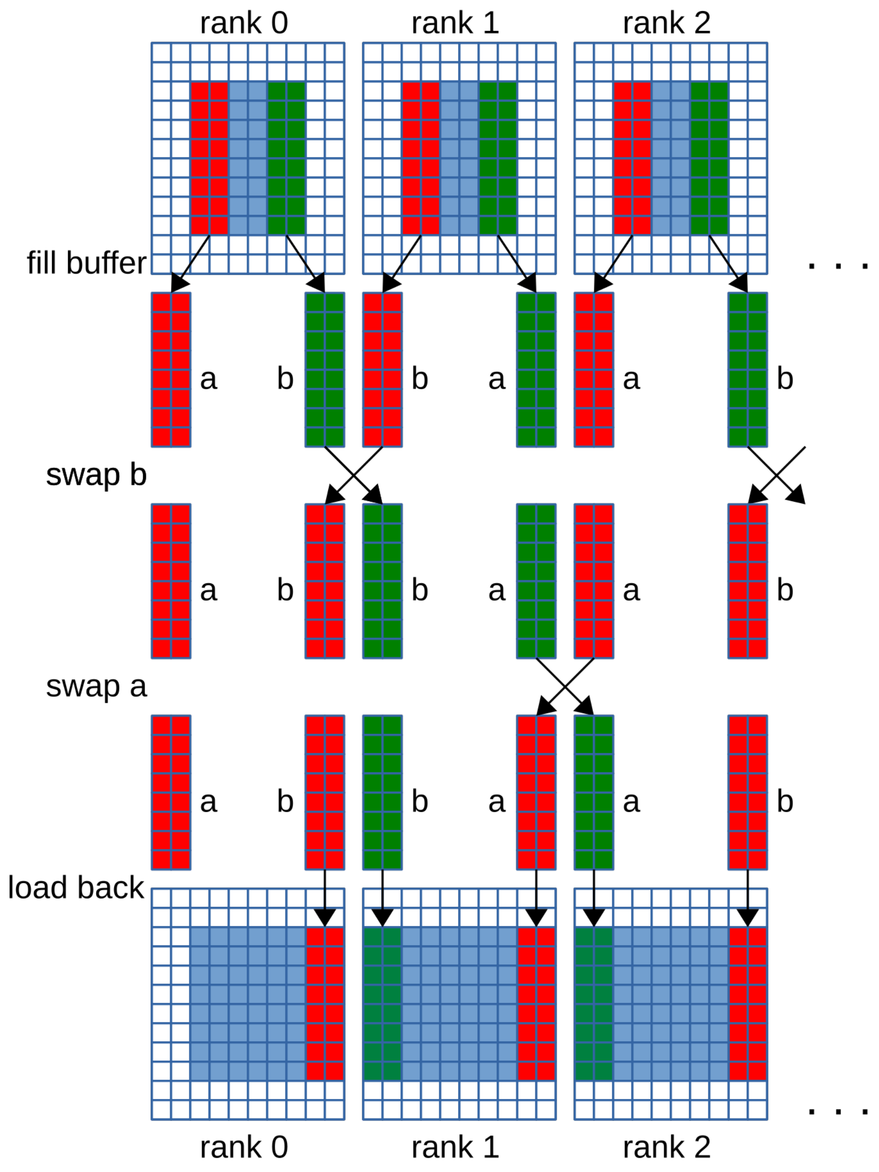
- This method will fail for very large grids that do not fit into the memory of a single node, as rank 0 receives all data from the other processes and needs to allocate a buffer of size nx_global * ny_global.

Parallelization

Boundary/ghost cells:

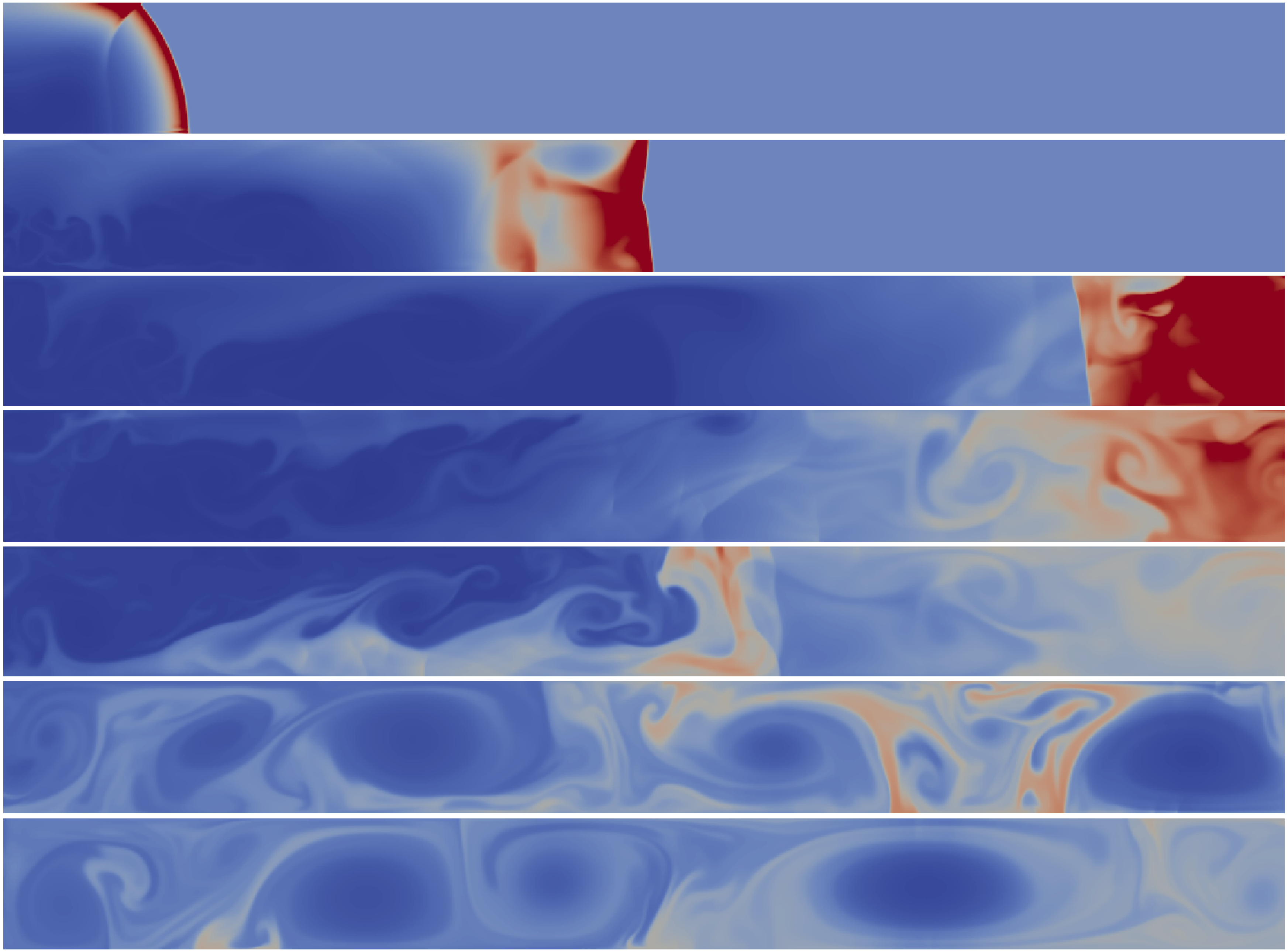
- Each process has its own ghost cells. To allow perturbations to flow from one grid segment to the other, we need to load the right border of process n-1 into the left ghost cells of process n, and vice versa for the left border. To account for this we wrote a new function mpi_sync that gets called at every timestep.
- mpi_sync does the following steps:
  - Initialize two buffers **a** & **b** of size 2 * ny * num_vars for each process.
  - For processes of even rank, load the left boundary into buffer **a** and right boundary into buffer **b**. For odd ranks, do the reverse.
  - Do a first synchronization where even ranks sync to the right and odd ranks sync to the left. This means that buffer **b** between neighboring ranks are switched via a call to the function

    MPI_Sendrecv_replace(buffer_b, buffer_size, …)
  - Do a second synchronization where odd ranks sync to the right and even ranks sync to the left, s.t. buffer **a** is switched between neighboring ranks.
  - Load the values stored in the buffers **a** & **b** into the ghost cells. For even ranks, buffer **b** now contains the values to be loaded into the ghost cells on the right, while buffer **a** contains the values to be loaded into the ghost cells on the left. For odd ranks the reverse is true.
  - Free buffers **a** & **b**.
- The layout of the buffers **a** & **b** allows for a very efficient message passing. Each process sends and receives data only once per timestep, and always half the processes are sending/receiving in parallel. This leads to a very small message passing overhead.



Visualization

This is a simulation on a 1000x100 grid running for around 75000 timesteps. We plot the density on a logarithmic scale.

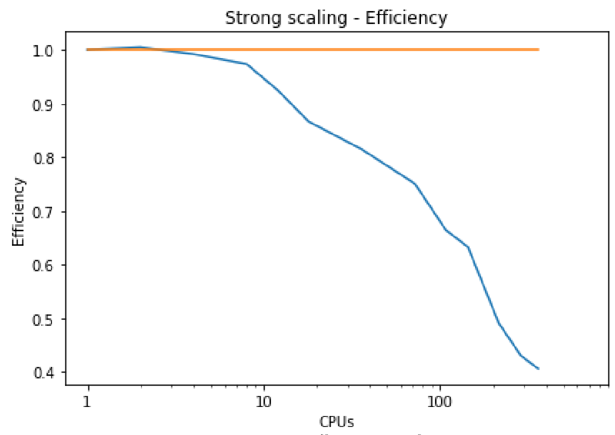


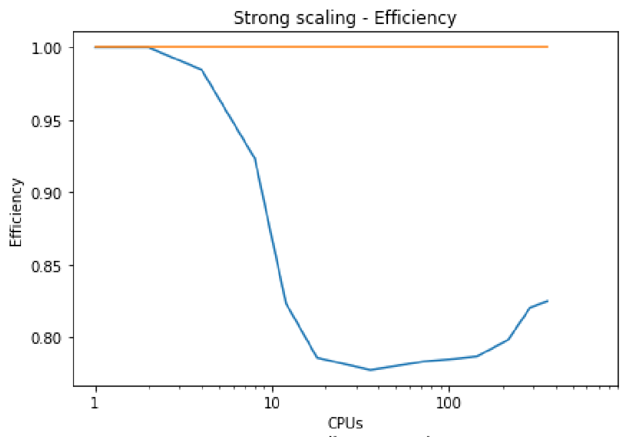
Scaling

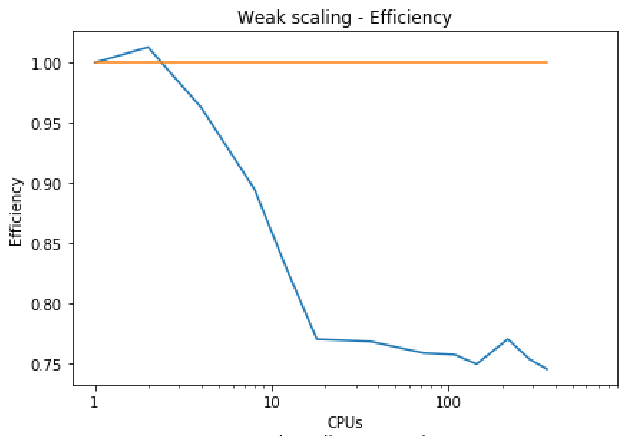
Gridsize: 2500x500

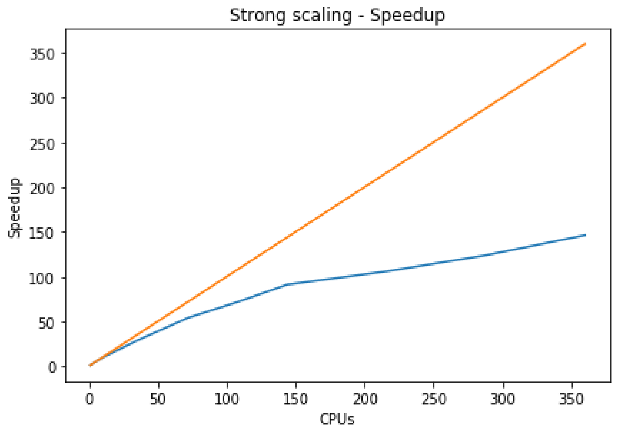
Gridsize: 100000x100

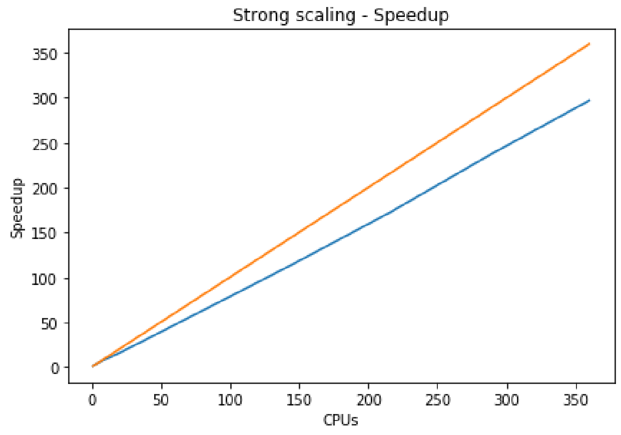
Gridsize per core: 250x250

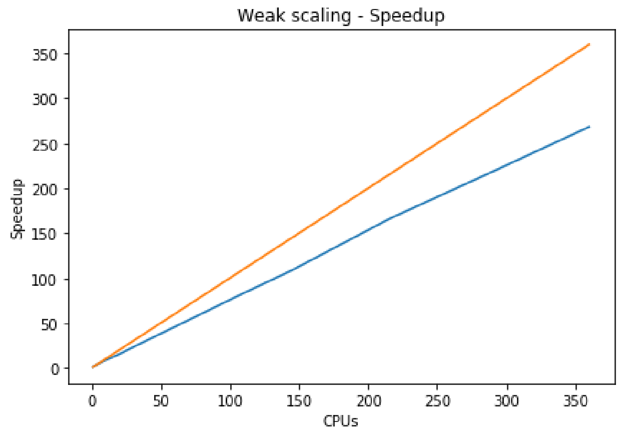












Strong scaling hits a plateau when the gridwidth of each process gets close to 0. In that case, some CPUs will not do any calculation and only send messages around. On a gridsize of 2500x500 this is already observable, as each process will have a width of around 2500 / 360 = 6-7, which is comparable to the width of the 4 ghost cells. More and more time will be spent on messaging and less on computing. On a very large grid of 100000x100 this is not observable, as even with 360 cores each process still has a gridsize of around 278x100. We observe a loss in efficiency up to 18 cores, which corresponds to one of the Intel Xeon E5-2695 v4 CPU of the XC40 compute nodes of Piz Daint. After that, the scaling is perfect, we even observe an increase in efficiency in higher core counts. This might be due to smaller grid sizes and thus better cache hits.

We observe the same loss of efficiency up to 18 cores in the weak scaling, after which the scaling is almost flat. The efficiency is above 75% in all cases.

All in all the scaling is quite well. The code is structured such that message passing is very efficient and very little time has to be spent sending data around. This allows for almost perfect scaling up to several hundreds cores.

Hybrid MPI/OpenMP hydro code

We also implemented a hybrid version of the hydro code using both MPI and OpenMP. In the scaling tests we saw that the MPI code scales perfectly across multiple nodes of Piz Daint, but suffers a loss of efficiency from 1 core to 18 cores. We hoped that OpenMP might increase performance when running on a single CPU.

The MPI part works as before, but now we parallelize the inner loops of the hydro_godunov function using OpenMP. In this function we iterate over all rows (or all columns) to solve the 1D Euler equations. Using OpenMP we create a parallel region around the hydro_godunov calls and parallelize the inner loops over rows and columns using

```
#pragma omp for schedule(dynamic, 2)
```

This way each thread will get a certain number of rows (columns) and compute only those. We choose a dynamic schedule with chunk size 2, which was determined to be the best performing after some tests.

One very important aspect of the OpenMP implementation is that each thread needs his own working space. During the calculations in hydro_godunov each thread needs its copy of the working variables to store temporary values. If we do not consider this, different threads will write to the same memory location and overwrite each others temporary values. We thus start the parallel region with

```
#pragma omp parallel private(Hw, Hvw)
```

This ensures that the working variables Hw & Hvw are private to each thread and thus allocated separately for each thread.

We observed that the hybrid MPI/OpenMP version is around 10-20% slower than the pure MPI version when tested on the same number of cores, in almost every case we considered. Finding the reasons behind this is quite difficult and we couldn’t get the hybrid version to run as fast as the MPI one, even though theoretically it should be possible. We suspect that there might be two reasons for that:

- The MPI version is already very efficient. Very little data needs to be send around and most of the time is spent computing. When profiling the code we found that the MPI syncing time was several orders of magnitude less than the computing time.
- Each OpenMP thread needs a separate working space. Thus a lot more memory has to be allocated and there might be some memory I/O performance hit.