Parallelizing The Hydro Code

Hannes Imboden, Philipp A. Huber

High Performance Computing

Final Project

June 2018

Contents

Hydro Code Explained

The hydro code is a simulation of a fluid or gas in an enclosed system. The initial conditions are set as to generate an unperturbed, homogeneous environment. Then, one can introduce points of higher or lower pressure or density to imitate an explosion or jet. The code then calculates the propagation of this perturbation using Godunov's method and outputs each frame as a single file, which can be used to create an animation.

There are five main components with different functionality to this Fortran based code:

- `main.f90`
  - calls `init_hydro` to initialize the grid with the initial conditions
  - executes the main time loop by calling
    - `output` to generate output data
    - `cmpdt` to compute the time step
    - `godunov` to calculate the fluid dynamics
- `module_hydro_principal.f90` contains the subroutines
  - `init_hydro`
  - `cmpdt`
  - `godunov` which calls
    - `make_boundary` to set the upper, lower, left and right boundaries of the grid
    - `allocate_work_space/deallocate_work_space` to (de)allocate space in the memory
    - other subroutines for calculation purposes
- `module_hydro_utils.f90` contains
  - `make_boundary`
  - other subroutines
- `module_hydro_IO.f90` contains the subroutines
  - `read_params` to read the parameters from the `input.nml` file
  - `output`
- `module_hydro_commun.f90` contains
  - `allocate_work_space/deallocate_work_space`
  - various other subroutines, mainly to define the variables used elsewhere in the code

Additionally, there are several input files to set the grid size with width nx and height ny and other initial conditions.

Parallelization Techniques

MPI

Introduction

MPI stands for *message passing interface.* This refers to the fact that parallel computing occupies multiple CPUs to fasten up the execution of a code. As these CPUs work independently and with their own associated memory, they will have to communicate in some way; this is what MPI does.
The advantage of MPI over other libraries is that one has much more control over how the code is parallelized. The programmer can explicitly choose how the different cores communicate to each other and which information they share.
On the other hand, this freedom makes parallelizing a code a much more complex problem than it could be with other libraires. It requires quite a deep understanding of how the code works to know where and what to adjust.

Parallelization Process

The main idea was to split the initial grid horizontally into equally sized parts in assigning each core a certain portion of it. Each process would then calculate the evolution of the fluid in his portion only. Finally, these different parts would be merged together to recreate the grid with its original size. This merging is where the main part of the paralellization process is happening:
For a process to be able to calculate a given pixel of its grid, it needs information of its four neighbouring pixels. Thus, to calculate the values of the lower and upper most pixels within its grid part, each process had to have access to the values the process below or above of it had calculated. For this purpose, so called *ghost cells* were introduced. Those are one-dimensional rows of pixels that only exist for calcuation purposes, but are not shown in the actual simulation.

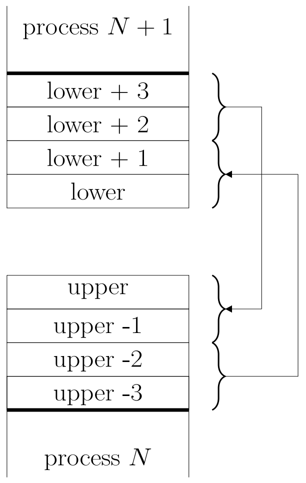

Figure 1: Communication

As illustrated in Figure 1, the thick lines are joined together in the actual simulation output. This behaviour could be achieved by letting the different processes communicate to each other in the following way:
First the odd processes send their values to the above and below processes while the even ones receive from below and above, respectively. Then, this procedure is repeated with the even ones sending and the odd ones receving. Special attention has to be paid to the lower most and the upper most process, because they do only have to send and

receive from one border. The main reason to implement the comunication in this way, was to avoid a deadlock. Other components of the code that needed to be adjusted were workspace allocation, time step computation and output formatting.

Results

To evaluate the performance of the parallelized code we performed a *strong* and a *weak scaling*. More precisely, for the strong scaling we ran the code with 1 to 36 cores and a *ny* of 9000 units and repeated this ten times to get a mean time per core. Similarly, for the weak scaling we varied the number of cores from 1 to 36, each time incrementing *ny* by 300 units to keep the grid per core constant. The results of the respective runtimes are displayed in Figure 2.

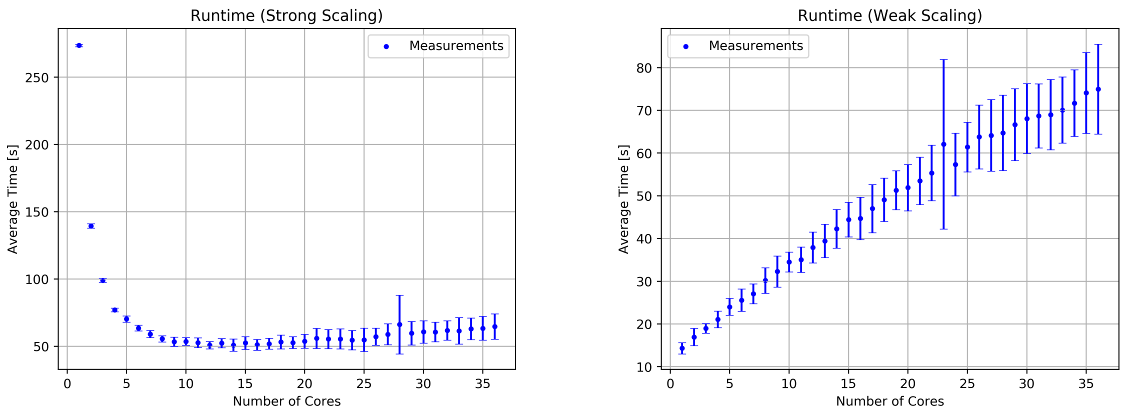

Figure 2: Respective runtimes for strong and weak scaling

To specify the percentage of the code that has actually been parallelized, we analyized the speedup (strong scaling) and the efficiency (weak scaling).
In case of the strong scaling, the speedup $s(N)$ is given by

$$s(N) = \frac{t_\mathrm{s}}{t_\mathrm{p}(N)} = \frac{1}{\alpha + \frac{1-\alpha}{N}}, \tag{1}$$

where t_s is the runtime of the serial code (i.e. one core), $t_\mathrm{p}(N)$ is the paralellized code as a function of the number of processes N and α is the non-parallelized fraction of the code. It is displayed in Figure 3.

The actual value for α can be obtained by fitting the function

$$f(N) = \frac{1}{\alpha + \frac{1-\alpha}{N}} \tag{2}$$

to the measurement points. In our case,

$$\alpha = 0.166 \pm 0.006. \tag{3}$$

This indicates that roughly 83.4% of the code is parallelized.

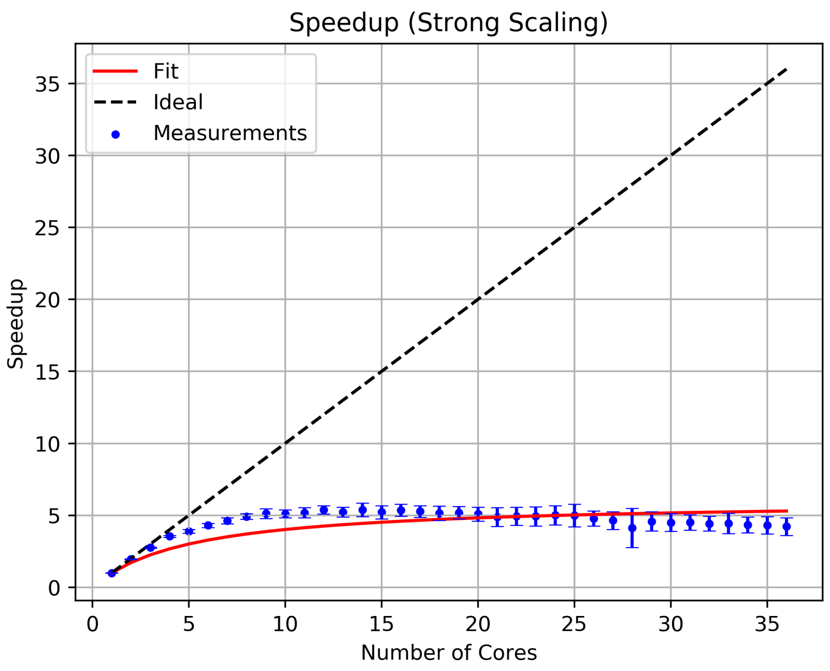

Figure 3: Speedup for strong scaling.

In case of the weak scaling, the efficiency ε is given by

$$\varepsilon(N) = \frac{t_\mathrm{s}}{t_\mathrm{p}(N)} = \frac{1}{\alpha \cdot N + (1-\alpha)}.$$

It is illustrated in Figure 4. Fitting the function to the measurements yields

$$\alpha = 0.145 \pm 0.002. \tag{4}$$

Again, this indicates that around 85.5% of the code is parallelized.

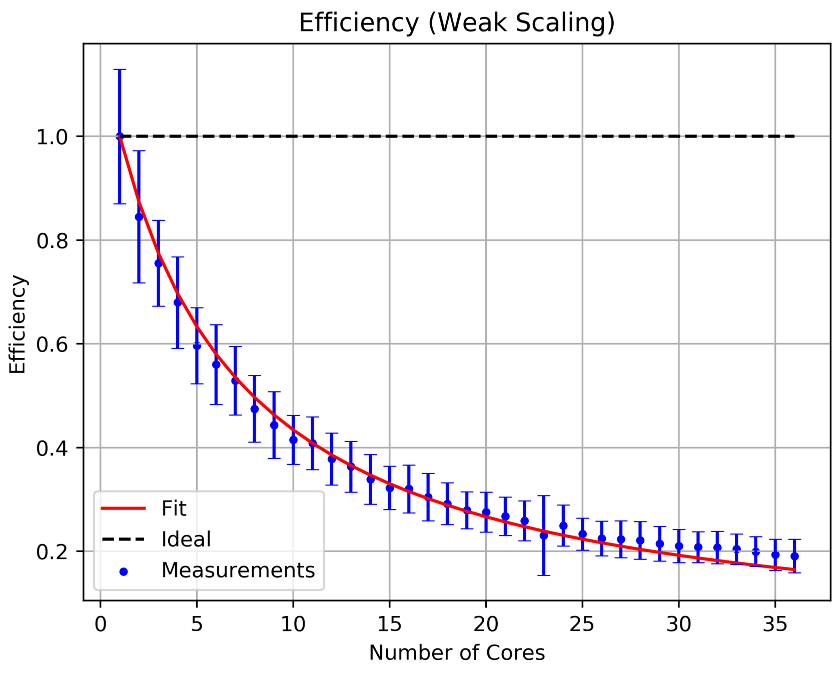

Figure 4: Efficiency for weak scaling.

Other Techniques

OpenMP

OpenMP stands for *open multiprocessing*. The fundamental difference between OpenMP and MPI is the parallelization process. MPI communicates between nodes, whereas with OpenMP the parallelization occurs within one node. The so-called *master thread* spawns several *slave threads* (this is called forking) to speed up a designated section of the code. The individual results are then recombined at the end of such a segment (which is called joining). This parallelization process is mostly used to speed up *for loops*.
What it may lack in flexibility (it can only run on shared-memory computers), it makes up in ease of implementation and communication speed.

MPI - OpenMP Hybrid

Implementing both of these methods can make use of the distributed memory architecture of a supercomputer, while further speeding up the runtime by occupying multiple threads. This combines the advantages of MPI and OpenMP.

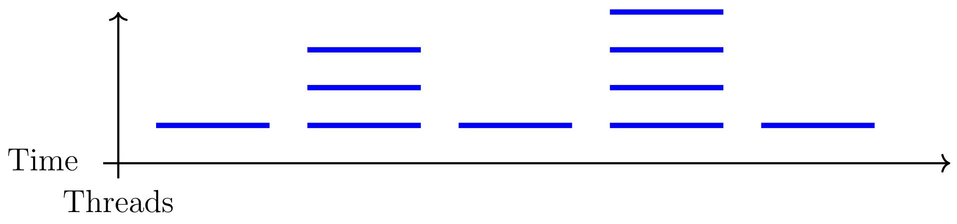

Figure 5: Operating principle of OpenMP

Discussion

As can be seen in Figure 2, the standard deviation on some of the runtimes can be quite big. This is mainly due that the whole run (going from 1 to 36 processes) was only performed 10 times. If one of these times is abnormally high, averaging over only 10 values will not suppress this spike well enough. This is what happened in our case. In the tenth iteration, one of the processes took double the time to complete its calculation. This may be due to a temporary performance drop at the Piz Daint supercomputer.

Our choice in implementing MPI relied on blocking communications. If one were to a non-blocking approach, the running time may be even reduced further, wich could increase the parallelized fraction of the code, α.
Although more difficult to implement, having a non-blocking implementation allows communication and computation to overlap, resulting generally in increased performance.

Conclusion