High Performance Computing 1b: Hydro Code

Mladen Ivkovic, Mischa Knabenhans, Rafael Souzalima

June 2016

The Hydro Code

The hydro code is a hydrodynamical simulation which solves the conservative Euler equations

$$\frac{\partial \vec{U}}{\partial t} + \vec{\nabla} \vec{F} = 0$$

for two dimensions, where $\vec{U} = (\rho, \rho u, \rho v, E)$ is the vector containing the conserved quantities density, momentum in x and y direction (u is the velocity in x direction and v the velocity in y direction) and energy; and $\vec{F} = (\rho u, \rho u^2 + p, (E + p)u)$ resp. $\vec{F} = (\rho v, \rho v^2 + p, (E + p)v)$ the flux function, where p is the pressure.

The simulation domain is rectangular, divided in square cells. The boundary conditions for the outermost cells are set to simulate a wall by mirroring the last two rows of cells before the wall.

It uses a Godunov's scheme with an exact Riemann solver, which yields the following numerical equation to be solved for each cell of the domain and for both dimensions separately:

$$U_{i,j}^{n+1} = U_{i,j}^{n} + \frac{\Delta t}{\Delta x} \left(F_{x,i+\frac{1}{2}}^{n+\frac{1}{2}} - F_{x,i-\frac{1}{2}}^{n+\frac{1}{2}} \right)$$

At each calculating step, the Godunov's scheme is solved first in one direction, then for the other. The direction in which will be calculated first switches every calculating step, leading to more precise results.

Parallelisation Strategy

The strategy used for the parallelisation of the code was to split the simulation domain between the processors, so each processor would only work on its own part of the domain, without having knowledge of what is happening in the domains of other processors. Before each step, the borders in the direction that is being calculated are communicated to neighbouring processors, creating a connection between the split domain parts.

In the initialisation phase of the code, first the best processor distribution for a given total number of processors and simulation domain size is calculated by choosing a combination of processors in x and y direction which minimises the communication time, considering the lattency and the message sizes.
Then, the domain is split amongst those processors as evenly as possible. If the grid size in x (resp. y) direction is not a multiple of the number of processors assigned to the x (resp. y) direction, the remaining amount of cells of this integer division is distributed one per processor, starting with the last in x (resp. y) direction, so that the processor ranked 0 would have less work, since it's the one responsible for runtime output.
After this step, every processor is "taught" who its neighbours are, or whether there is a wall next to it.

The advantage of this parallelisation is that any domain size can be calculated for any numbers of processors and the most expensive part of the work is done in parallel, but the processor distribution along a direction is not freely choosable, even though it may be enforced by setting the simulation domain sizes appropriately.

Bottlenecks

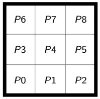
We identified the bottlenecks of the code to be in the communication between the processors.

On one hand, on each calculation step, the maximal allowed time step, which ensures conservation of the conservative variables, must be recalculated by each processor. Then the minimal value of this maximal timestep must be communicated to all processors for the simulation to evolve at the same pace for each part of the domain. For this communication, we used the MPI_ALLREDUCE routine. This is a blocking call and might waste processing time if processors have to wait for eachother often. To minimise waiting time, we tried to distribute the work as evenly as possible amongst the processors.

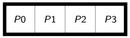
On the other hand, before each calculation step, the borders of the domain have to be communicated by each processor to each of its neighbours. For this, we used a MPI_SENDRECV routine, which is a blocking call as well. Since we're sending a part of an array for every communication, the MPI routine will create an array temporary for each communication. This bottleneck might be optimised by using nonblocking calls and defining a new MPI structure type, but we couldn't realise those ideas due to time limitations.

Performance and Scaling: Weak Scaling

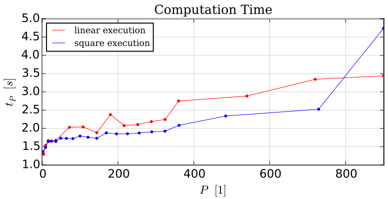
Since our parallelisation allows domain splitting in both directions, we wanted to create a comparison between a strictly "linear execution", where the domain is split only in one direction, and a "square execution", where the number of processors assigned to each direction is equal.

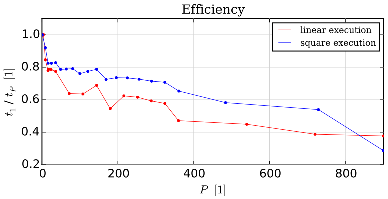


Processor distribution for a "square execution"



Processor distribution for a "linear execution"



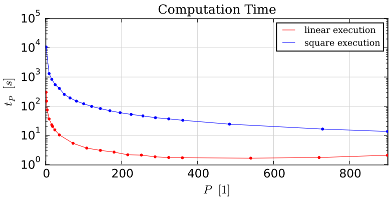


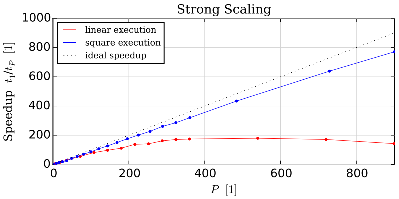
P: Number of processors used. t_P : Measured execution time for P processors.

For the weak scaling measurements, all processors were assigned the same workload of 200×200 cells for 100 calculation steps, so the total number of cells increases with the number of processors P ranging from 1 to 900.

Surprisingly the square execution requires less computation time, even though a square processor distribution requires more communications between processors, while the amount of communicated data is constant. Since the number of cells per processor remain constant for all P, we may assume that the boundary communication time remains constant for all P as well (at least for $P \geq 3$, when there is at least 1 processor with the maximal amount of neighbours.) Therefore, with increasing P, the timestep communication (which uses the MPI_ALLREDUCE routine) should dominate the efficiency loss.

Performance and Scaling: Strong Scaling





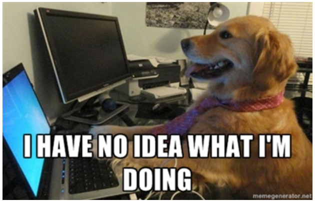
P: Number of processors used. t_P : Measured execution time for P processors.

For the strong scaling performance test, a fixed grid size was chosen and calculated for 100 calculation steps and computed with a different amount of processors P ranging from 1 to 900.
The grid size for the linear execution was 45360×200 and for the square execution it was 16200×16200.
Please note that therefore the computation times in the plot above cannot be compared directly.

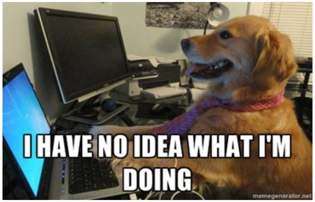
The square execution shows much better scaling than the linear execution. This can be explained by looking at how the boundary communication time changes with the number of processors used: while we may assume the total lattency effects to be constant for all numbers of processors used, the transfer time ("talking time") reduces with the number of processors for a fixed total size grid because less and less cells need to be communicated between two processors. It can be shown (see appendix) that the rate of change of the boundary communication time for a whole calculation step t_c with respect to the number of processors used P is $\frac{dt_{c,lin}}{dP} \propto -P^{-2}$ for the linear execution and $\frac{dt_{c,sq}}{dP} \propto -P^{-3/2}$ for the square execution. So the communication time for the square execution will decrease faster with increasing number of used processors, leading to the better scaling property for the square execution.

High Quality Output

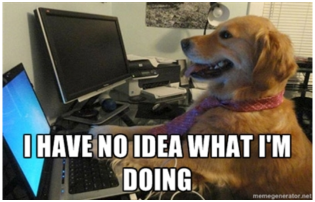
Below are three high quality outputs at calculation steps X, Y and Z for the gridsize $A \times B$, calculated on n processors.



Output X



Output X



Output X

Appendix: Estimating the Boundary Communication Time per Calculation Step

We want to estimate the communication time for the boundaries per calculation step of the hydro code. It is of special interest to us because we expect the timestep communication to remain constant for all calculation steps for a given number of processors used P.

We assume that during the communication phase of the code at each moment a processor communicates with exactly 1 neighbour and the processors communicate exclusively with their neighbours. For the linear execution, the number of neighbours n_n each processor has is $n_n = 2$, while for the square execution it is $n_n = 4$. We don't consider the processors with domain parts on the edge of the total domain (which are not communicated, instead a wall is created), since the total communication time will be determined by the slowest process, in this case the one with the most communications.

Now we can express the communication time as $t_c = n_n \cdot (t_x + t_y)$, where t_x and t_y are the communication times in x resp. y direction.
$t_{x,y}$ can be expressed as a sum of two processes: the lattency l that is needed to establish the communication and the actual transfer time. We express the actual transfer time as the product of the transfer time per unit that shall be sent, which will be s/c, the unit size divided by the transfer speed, and the number of units to be transferred. In the case of the hydro code, this number will be $2 \cdot 4 \cdot \Delta x = 8\Delta x$ resp. $8\Delta y$, because we need to send 2 rows resp. columns, each containing 8 conserved quantities, for the whole column resp. row. This gives us $t_x = l + 8\frac{s}{c}\Delta y$ and $t_y = l + 8\frac{s}{c}\Delta x$. Note that for the linear execution, $t_y = 0$ because there are no communications in the y direction, and for the square execution: $\Delta x = \Delta y$.
This gives us:

$$t_{c,lin} = n_n \cdot (l + 8\tfrac{s}{c}\Delta y) = 2 \cdot (l + 8\tfrac{s}{c}\Delta y) \qquad \text{for linear execution}$$

$$t_{c,sq} = n_n \cdot \left(2l + 8\tfrac{s}{c}(\Delta x + \Delta y)\right) = 8 \cdot (l + 8\tfrac{s}{c}\Delta y) \qquad \text{for square execution}$$

In a weak scaling measurement, all the parameters remain constant for all P, the number of processors used. Thus we may conclude that the boundary communication time will remain constant in this case for all P. Instead, the timestep communication time will increase with higher P, slowing the code down and explaining why the computation time rises.
With our assumptions, we see that in the case of weak scaling, $t_{c,sq} = 4t_{c,lin}$ and we expect the square execution time to be higher for all P than the linear execution time. Measurements showed (see page 5) that this is not the case. We can't explain that behaviour.

In a strong scaling measurement, where the total domain size is fixed for all P, the row and column lenght will vary with the number of processors used. We can say that for a linear execution $\Delta y = \frac{ny}{P}$, where ny is the length of the total domain in y direction. For a square execution, it is $\Delta y = \frac{ny}{\sqrt{P}}$.
This leaves us with:

$$t_{c,lin} = 2 \cdot (l + 8\tfrac{s}{c}\tfrac{ny}{P}) \Rightarrow \frac{dt_{c,lin}}{dP} = -16\frac{s}{c}\frac{ny}{P^2} \qquad t_{c,sq} = 8 \cdot (l + 8\tfrac{s}{c}\tfrac{ny}{\sqrt{P}}) \Rightarrow \frac{dt_{c,sq}}{dP} = -32\frac{s}{c}\frac{ny}{P^{3/2}}$$

We see that the rate of change of the boundary communication time for a whole calculation step t_c with respect to the number of processors used P in strong scaling measurements will decrease both for the linear and the square execution, but it will decrease for the square execution much faster, causing better speedups.