Technische Universität München
Lehrstuhl Informatik V
M. Bader, O. Meister

WS 2013/14
Worksheet 5
January, 8th 2014

Tutorial: HPC - Algorithms and Applications WS 13/14

Complete the following assignments (alone or in a group), and send your source code via e-mail to `meistero@in.tum.de` until Sunday, January, 19th 2014.

Worksheet 5: Shallow Water Equations in CUDA

Assignment 1: Case Study

Read the case study on this worksheet. Keep in mind that it is not necessary for you to understand every detail, only the general concept.

a) Answer the following questions on model and discretization of the Shallow Water Equations:

   i) What is a flux solver? Which data does it operate on?

   ii) What are the source terms in the shallow water equations?

b) Answer the following questions on implementation of the SWE code:

   i) What components does a block consist of and how do they differ?

   ii) What data dependency exists between `computeFluxes()` and `eulerTimestep()`?

Assignment 2: Make it run

Download the exercise code and run it on your system.
Linux: Installation instructions:

a) Install required libraries (if necessary).
   Ubuntu: `sudo apt-get install libxi-dev libxmu-dev libsdl1.2-dev libsdl1.2-debian scons`

b) Download and extract exercise files

c) To compile, change to the SWE directory and type `scons`.

d) If CUDA is not found, you can set the CUDA folder manually in `./SWE_gnu_cuda.py` and call `scons buildVariablesFile=./SWE_gnu_cuda.py` instead.

e) You can use build variable files to modify other parameters as well, for example to enable OpenGL output. Look in `./build/options` for a few examples.

f) The executable is generated in `./build`.

Windows: A Visual Studio project is contained in the files.

Assignment 3: Net updates kernel

Implement the net updates kernel in `src/SWE_WavePropagationBlockCuda_kernels.cu` and run the program (default: OpenGL output)

a) Inside `computeNetUpdatesKernel`, compute the thread local cell indices and the index in the cell data arrays (as done in `updateUnknownsKernel`). For the vertical edge on the left of the cell and the horizontal edge on the bottom of the cell:

   i) Determine array positions of the cell data for both cells next to the edge

   ii) Call `fWaveComputeNetUpdates` (see `src/solvers/FWaveCuda.h` for the function signature) and read the correct arguments from the data arrays.

   iii) Write the net updates back to the correct positions in the global net update arrays.

   iv) Update the maximum wave speed.

   Do not forget to synchronize where necessary!
   Note: You can refer to the C++ version of the kernel in `src/SWE_WavePropagationBlock.cpp`.

b) Run the program. If your solution is correct, the visual output should produce plausible behaviour for all test cases (press 1, 2, 3).

   i) Radial wave that propagates from the center to the boundary.

   ii) Radial wave with variable sea floor, similar result to i).

   iii) Diagonal wave that travels back and forth between bottom left and top right corner.

Case Study: Shallow Water Equations

Part 1: Model and numerics

In the following case study, we consider the so-called *shallow water equations*. They describe the behaviour of a fluid, in particular water, of a certain (possibly varying) depth h in a two-dimensional domain – imagine, for example, a puddle of water or a shallow pond (and compare the 1D sketch given in Figure 1). The main modelling assumption is that we can neglect effects of flow in vertical direction. The resulting model therefore proves to be useful in surprisingly many situations. The simulation of tsunamis, for example, can be efficiently done using shallow water equations (with appropriate extensions). While an ocean can hardly be considered as "shallow" in the usual sense, tsunami waves (in contrast to regular waves induced by wind, e.g.) affect the entire water column, such that effects of vertical flow can again be neglected. To allow for a non-even sea bottom (as required for accurate modelling of tsunamis), we include the elevation b of the sea floor in our model (compare Figure 1).

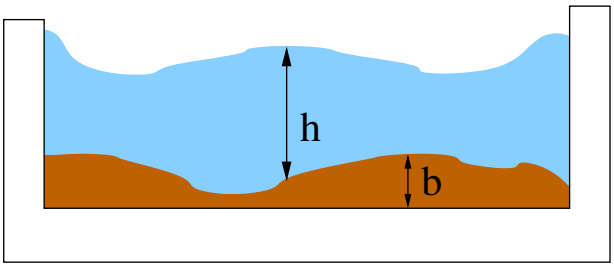

Figure 1: Water level h and bottom level b, as used in the presented shallow water model.

The shallow water equations describe the changes of water depth h and horizontal velocities v_x and v_y (in the resp. coordinate directions) over time, depending on some initial conditions – in the case of tsunami simulation, these initial conditions could, for example, result from an initial elevation of the sea floor caused by an earthquake. The respective changes in time can be described via a system of partial differential equations:

$$\begin{aligned}
\frac{\partial h}{\partial t} + \frac{\partial (v_x h)}{\partial x} + \frac{\partial (v_y h)}{\partial y} &= 0 \\
\frac{\partial (h v_x)}{\partial t} + \frac{\partial (h v_x v_x)}{\partial x} + \frac{\partial (h v_y v_x)}{\partial y} + \frac{1}{2} g \frac{\partial (h^2)}{\partial x} &= -gh \frac{\partial b}{\partial x}, \\
\frac{\partial (h v_y)}{\partial t} + \frac{\partial (h v_x v_y)}{\partial x} + \frac{\partial (h v_y v_y)}{\partial y} + \frac{1}{2} g \frac{\partial (h^2)}{\partial y} &= -gh \frac{\partial b}{\partial y},
\end{aligned} \tag{1}$$

The equation for h is obtained, if we examine the conservation of mass in a control volume. The equations for hv_x and hv_y result from conservation of momentum (note that h is directly related to the volume, and thus the mass of the water – thus hv_x can be interpreted as a momentum).

The terms $\frac{1}{2} g \frac{\partial (h^2)}{\partial x}$ and $\frac{1}{2} g \frac{\partial (h^2)}{\partial y}$ model a gravity-induced force (g being the constant for the gravitational acceleration, $g = 9.81 \mathrm{ms}^{-2}$), which results from the hydrostatic pressure. The right-hand-side source terms $-gh \frac{\partial b}{\partial x}$ and $-gh \frac{\partial b}{\partial y}$ model the effect of an uneven ocean floor (b obtained from respective bathymetry data).

A Discrete Finite Volume Model

The system of equations (1) is usually too difficult to be solved exactly – hence, we will derive a simplified discrete model as an approximation. At the same time, this will provide us with a better understanding of what effects are modelled by each of the terms (and how). First, we assume that our unknown functions $h(t,x,y)$, $q_x(t,x,y) := h(t,x,y)v_x(t,x,y)$, $q_y(t,x,y) := h(t,x,y)v_y(t,x,y)$, as well as the given bottom level $b(x,y)$, are approximated on a Cartesian mesh of grid cells, such as illustrated in Figure 2. In each grid cell, with indices (i,j), we assume that the unknowns have constant values h_{ij}, p_{ij}, q_{ij}, and b_{ij} (see also Figure 3) – using p as variable for the products hv_x, and q for hv_y.

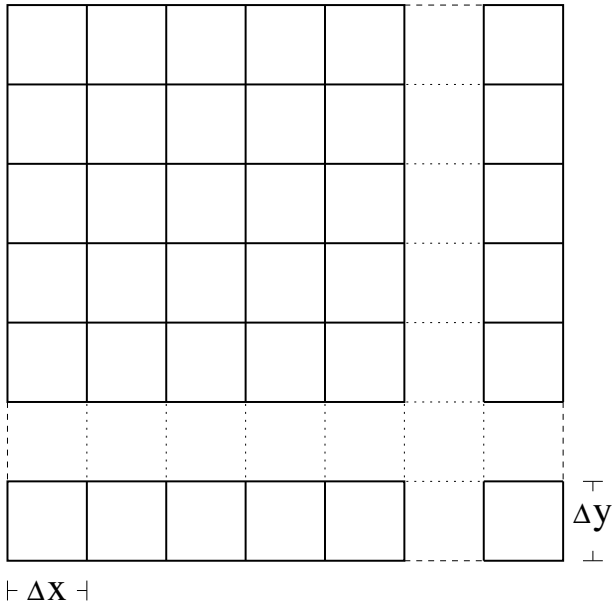

Figure 2: Cartesian grid for the described Finite Volume model.

We can then formulate our system (1) for each cell:

$$\begin{aligned}
\left.\frac{\partial h}{\partial t}\right|_{ij} + \left.\frac{\partial p}{\partial x}\right|_{ij} + \left.\frac{\partial q}{\partial y}\right|_{ij} &= 0 \\
\left.\frac{\partial p}{\partial t}\right|_{ij} + \left.\frac{\partial (v_x p)}{\partial x}\right|_{ij} + \left.\frac{\partial (v_y p)}{\partial y}\right|_{ij} + \left.\frac{1}{2} g \frac{\partial (h^2)}{\partial x}\right|_{ij} &= -gh \left.\frac{\partial b}{\partial x}\right|_{ij}, \\
\left.\frac{\partial q}{\partial t}\right|_{ij} + \left.\frac{\partial (v_x q)}{\partial x}\right|_{ij} + \left.\frac{\partial (v_y q)}{\partial y}\right|_{ij} + \left.\frac{1}{2} g \frac{\partial (h^2)}{\partial y}\right|_{ij} &= -gh \left.\frac{\partial b}{\partial y}\right|_{ij}.
\end{aligned} \tag{2}$$

Note that the values in each cell are still time-dependent in this equation! Hence, we further simplify the situation in the sense that we only consider the unknowns at certain time steps $t_n := n\tau$. We'll write the unknowns in a cell (i,j) at time step n as $h_{ij}^{(n)}$, $p_{ij}^{(n)}$, and $q_{ij}^{(n)}$ (but, for simplicity, leave the b_{ij} constant in time). We can now approximate the time derivatives in equation (2) as

$$\left.\frac{\partial h}{\partial t}\right|_{ij}(t_n) \approx \frac{h_{ij}^{(n+1)} - h_{ij}^{(n)}}{t_{n+1} - t_n} = \frac{h_{ij}^{(n+1)} - h_{ij}^{(n)}}{\tau}$$

(similar for p_{ij} and q_{ij}), and evaluate all remaining terms in the equation at time t_n, which leads to a forward Euler scheme:

$$\begin{aligned}
&\frac{h_{ij}^{(n+1)} - h_{ij}^{(n)}}{\tau} + \left.\frac{\partial p}{\partial x}\right|_{ij}^{(n)} + \left.\frac{\partial q}{\partial y}\right|_{ij}^{(n)} = 0 \\
&\frac{p_{ij}^{(n+1)} - p_{ij}^{(n)}}{\tau} + \left.\frac{\partial (v_x p)}{\partial x}\right|_{ij}^{(n)} + \left.\frac{\partial (v_y p)}{\partial y}\right|_{ij}^{(n)} + \frac{1}{2} g \left.\frac{\partial (h^2)}{\partial x}\right|_{ij}^{(n)} = -gh \left.\frac{\partial b}{\partial x}\right|_{ij}, \\
&\frac{q_{ij}^{(n+1)} - q_{ij}^{(n)}}{\tau} + \left.\frac{\partial (v_x q)}{\partial x}\right|_{ij}^{(n)} + \left.\frac{\partial (v_y q)}{\partial y}\right|_{ij}^{(n)} + \frac{1}{2} g \left.\frac{\partial (h^2)}{\partial y}\right|_{ij}^{(n)} = -gh \left.\frac{\partial b}{\partial y}\right|_{ij}.
\end{aligned} \tag{3}$$

If we solve the equations for $h_{ij}^{(n+1)}$, $p_{ij}^{(n+1)}$, and $q_{ij}^{(n+1)}$, respectively, we obtain an explicit scheme to compute the unknowns of time step t_{n+1} from the known values of time step t_n (starting with given initial values at t_0).

Computing the Spatial Derivatives – Fluxes at Edges

Recall that the term $\left.\frac{\partial p}{\partial x}\right|_{ij}^{(n)}$ means that we should compute the value of the derivative $\frac{\partial p}{\partial x}$ in cell (i, j) at time step n. Assume that we were able to compute the values of p at the boundary of the cells, which we denote as $p|_{i-\frac{1}{2},j}^{(n)}$ for the value on the left boundary, and $p|_{i+\frac{1}{2},j}^{(n)}$ for the value on the right boundary – see also the illustration in Figure 3. The partial derivatives could then be approximated as

$$\left.\frac{\partial p}{\partial x}\right|_{ij}^{(n)} \approx \frac{p|_{i+\frac{1}{2},j}^{(n)} - p|_{i-\frac{1}{2},j}^{(n)}}{\Delta x} \quad \text{and} \quad \left.\frac{\partial q}{\partial y}\right|_{ij}^{(n)} \approx \frac{q|_{i,j+\frac{1}{2}}^{(n)} - q|_{i,j-\frac{1}{2}}^{(n)}}{\Delta y}, \tag{4}$$

where Δx and Δy are the cell sizes in x- and y-direction.

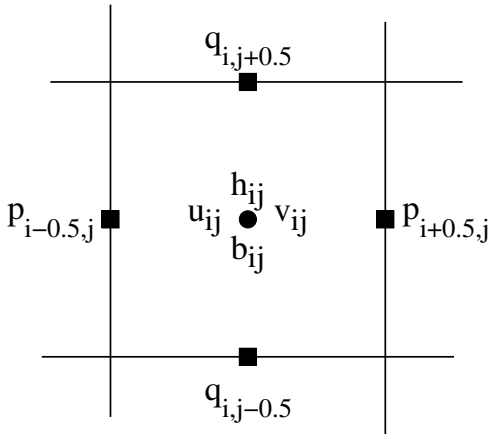

Figure 3: Cell-centred unknowns and fluxes on the cell edges.

Taking averages on Cell Edges

As we usually store our unknowns for each cell, there is no immediate approximation of the values $p|_{i-\frac{1}{2},j}^{(n)}$ or $q|_{i,j+\frac{1}{2}}^{(n)}$, as they are situated right on the boundary between two cells. A simple way to

compute an approximate value would be to compute the average:

$$p|^{(n)}_{i+\frac{1}{2},j} = \frac{1}{2}\left(p^{(n)}_{ij} + p^{(n)}_{i+1,j}\right) \quad \text{or} \quad q|^{(n)}_{i,j-\frac{1}{2}} = \frac{1}{2}\left(q^{(n)}_{i,j-1} + q^{(n)}_{ij}\right) \tag{5}$$

If we apply this method to all respective terms in (3), we can – in principle – compute all new unknowns for time step t_{n+1}.

Physical Interpretation: Computation of Fluxes across Cell Boundaries

Let's take a short look at the interpretation of the terms $p = v_x h$ and $q = v_y h$: at the boundaries $i \pm \frac{1}{2}$, v_x denotes the velocity of the fluid across the cell boundary; similar, v_y is the velocity across the boundaries $j \pm \frac{1}{2}$. h denotes the water depth, and thus basically the mass of the water situated at the boundary. The product $v_x h$ thus reflects the amount of water that is transported across the cell boundary per time. The term

$$\left.\frac{\partial p}{\partial x}\right|^{(n)}_{ij} + \left.\frac{\partial q}{\partial y}\right|^{(n)}_{ij} \approx \frac{(v_x h)|^{(n)}_{i+\frac{1}{2},j} - (v_x h)|^{(n)}_{i-\frac{1}{2},j}}{\Delta x} + \frac{(v_y h)|^{(n)}_{i,j+\frac{1}{2}} - (v_y h)|^{(n)}_{i,j-\frac{1}{2}}}{\Delta y} \tag{6}$$

thus determines the net in- or outflow of water into (or out of) the cell. As we see from equation (3), this term is used to determine the change of the water height: if additional mass is flowing into a cell, the water level h will rise – or fall, if the balance of inflow/outflow is negative.

Hence, we can now give an interpretation of our equation to compute h: in each time step we use the current velocities to determine so-called *flux terms* at all boundaries; the balance of these fluxes determines the water in- and outflow and, as such, the change of the water level in each cell.

For the two momentum equations in (3), we have to compute fluxes in a similar way – however, the physical interpretation of these flux terms is no longer as obvious as for the h-equation.

Lax-Friedrichs Flux Computation

It turns out that computing the flux as a simple average of the values in adjacent cells, as in equation (5), will lead to an unstable numerical method: the discrete approximation leads to spurious, physically incorrect solutions, which can be further and further amplified and lead to diverging solutions. One possibility to avoid these problem, is to introduce certain correction terms. For example, the fluxes at the boundaries can be computed according to the Lax-Friedrichs method:

$$p|^{(n)}_{i+\frac{1}{2},j} = \frac{1}{2}\left(p^{(n)}_{ij} + p^{(n)}_{i+1,j}\right) + \frac{\Delta x}{2\tau}\left(h^{(n)}_{ij} - h^{(n)}_{i+1,j}\right) \tag{7}$$

This can be interpreted as a certain damping of the solution (due to friction, e.g.), which is sufficient to stabilise the numerical solution. (Hint: computing the difference of the terms $p|^{(n)}_{i+\frac{1}{2},j}$ and $p|^{(n)}_{i-\frac{1}{2},j}$ will lead to a discretised second derivative in the h terms – which is equivalent to a diffusion term.) Note that we also need to use this different flux computation scheme for the momentum equations.

Discretisation of the Source Terms

Special care is also required for the computation of the source terms $-gh\left.\frac{\partial b}{\partial x}\right|_{ij}$ and $-gh\left.\frac{\partial b}{\partial y}\right|_{ij}$. Assume that our ocean is "at rest", i.e. all velocities are zero and thus $p = 0$ and $q = 0$. A situation

where $h_{ij} + b_{ij} = \text{const}$, i.e., where the sea surface is constant, should then be in equilibrium and stay constant as solution. From equation (3), we see that the equalities

$$\frac{1}{2} g \frac{\partial (h^2)}{\partial x}\bigg|_{ij}^{(n)} = -gh \left.\frac{\partial b}{\partial x}\right|_{ij} \quad \text{and} \quad \frac{1}{2} g \frac{\partial (h^2)}{\partial y}\bigg|_{ij}^{(n)} = -gh \left.\frac{\partial b}{\partial y}\right|_{ij}. \tag{8}$$

have to be satisfied exactly, also in their discretised form. As an exercise, you may check the following two "candidates" for discretisation of the right-hand side:

$$h \left.\frac{\partial b}{\partial x}\right|_{ij} = h_{ij} \frac{b_{i+1,j} - b_{i-1,j}}{2\Delta x} \quad \text{vs.} \quad h \left.\frac{\partial b}{\partial x}\right|_{ij} = \frac{h_{i+1,j} + h_{i-1,j}}{2} \cdot \frac{b_{i+1,j} - b_{i-1,j}}{2\Delta x}. \tag{9}$$

It turns out that only the right-hand form of the discretisation satisfies equation (8).

Further Comments

The model and the numerical scheme used for our project are strongly simplified. However, we can use the resulting algorithm for the simulation of simple scenarios: a basin where water is splashing back and forth, or an elevated water region that collapses, causing a shock waves that propagates in all directions. A description of the implementation of this model will be presented on worksheet 2 (together with some further numerical features).

In general, our numerical scheme can be interpreted as a *Finite Volume* discretisation of the shallow water equations given in (1). In particular, the approach of studying inflow and outflow of physical quantities in certain control volumes (here, the cells of our Cartesian mesh), are typical for Finite Volume models. Vice versa, a discrete Finite Volume scheme, as we obtain for our numerical scheme will turn into the system of partial differential equations (1), if the mesh size $\Delta x \to 0$. A thorough introduction to Finite Volume methods, especially for equations of the same type as the shallow water equations, is given in the textbook of LeVeque [1], for example.

Shore Regions – Wetting and Drying

For full-featured tsunami simulations, inundation of shore regions is a particular challenge. In dry coastal regions, we can simple set the water depth to $h = 0$, but the computation of the velocities $v_x = p/h$ and $v_y = q/h$ becomes delicate. Much more difficult is the treatment of the conditions defined in equation (8), and the possibility that cells changed between wet ($h > 0$) and dry ($h \leq 0$) states. The implementation provided with this worksheet attempts an ad-hoc fix to this problem, which works for some simple scenarios, but is far from being a robust solution – feel free to check out more difficult scenarios and examine the problem that occur. You are more than welcome to try to improve the present approach (and report your findings)!

References

[1] R. J. LeVeque: *Finite Volume Methods for Hyperbolic Problems*, Cambridge Texts in Applied Mathematics, 2002

Part 2: Implementation

In part 1, we have discussed the shallow water equations as a model for certain fluid flow problems, and introduced a simple numerical scheme to solve the time-dependent equations. Part 2 will discuss an example for a C/C++ implementation of the numerical algorithm, and will also present some aspects that were left out in worksheet 1 – especially the treatment of domain boundaries.

Data Structure: Class `SWE_Block`

For the simulation of the shallow water model, we require a regular Cartesian grid, where each grid cell carries three unknowns – the water level h and momentum p and q (in x- and y- direction, resp.). Hence, our main data structures are three two-dimensional arrays `h`, `u`, and `v` (names chosen for historical reasons), which will hold the unknowns. A fourth array `b` will hold the bottom levels b_{ij}. To implement the behaviour of the fluid at boundaries (see next section), we will add an additional layer of so-called *ghost cells* to our actual Cartesian grid, as illustrated in Figure 4.

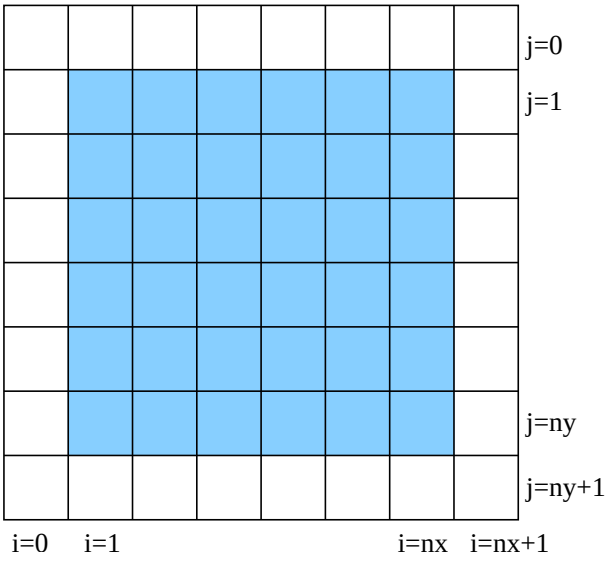

Figure 4: Cartesian grid block with a layer of ghost cells, as used in the class `SWE_Block`

Hence, for `i=1...nx` and `j=1...ny`, the unknowns `h[i][j]`, `u[i][j]`, and `v[i][j]` represent the unknowns in the fluid domain. For `i=0`, `j=0`, `i=nx+1`, and `j=ny+1`, we will use the respective variables to prescribe boundary conditions at the left, bottom, right, and top boundary, respectively.

We combine the arrays `h`, `u`, and `v` into a class `SWE_Block`, which will, in addition, contain the following static and member variables:

- six two-dimensional arrays `Fh`, `Fu`, `Fv`, and `Gh`, `Gu`, `Gv` as helper variables to compute the fluxes across cell boundaries. The `F`-variables will hold fluxes across left/right boundaries; the `G`-variables fluxes across bottom/top boundaries.
- `nx` and `ny` hold the size of the Cartesian grid block (see Figure 4).
- `dx` and `dy` hold the size of the grid cells.

- `dt` determines the size of the time step.

Treatment of Boundaries

In each time step, our numerical algorithm will compute the flux terms for each edge of the computational domain. To compute the fluxes, we require the values of the unknowns in both adjacent cells. At the boundaries of the fluid domain, the ghost layer makes sure that we also have two adjacent cells for the cell edges on the domain boundary. The values in the ghost layer cells will be set to values depending on the values in the adjacent fluid domain. We will model three different situations:

Outflow: `h`, `u`, and `v` in the ghost cell are set to the same value as in the adjacent fluid cell. This models the situation that the unknowns do not change across the domain boundary (undisturbed outflow).

Wall: At a wall, the velocity component normal to the boundary should be 0, such that no fluid can cross the boundary. To model this case, we set the normal velocity, e.g. `u[0]` at the left boundary, to the negative value of the adjacent cell: `-u[1]`. The interpolated value at the boundary edge will then be 0 (`h` is identical in both cells due to the imposed boundary condition). The other two variables are set in the same way as for the outflow situation.

Connect: With the connect case, we can connect a domain at two boundaries. If we connect the left and right boundary, we will obtain a periodically repeated domain. Here, all ghost values are determined by the values of the unknowns in the fluid cell adjacent to the connected boundary.

To implement the boundary conditions, the class `SWE_Block` contains an array of four enum variables, `boundary[4]` (for left/right/bottom/top boundary), that can take the values `OUTFLOW`, `WALL`, and `CONNECT`.

Multiple Blocks

Via the connect boundary condition, it is also possible to connect several Cartesian grid blocks to build a more complicated domain. Figure 5 illustrates the exchange of ghost values for two connected blocks.

To store the neighbour block in case of a `CONNECT` boundary, `SWE_Block` contains a further array of four pointers, `neighbour[4]` (for left/right/bottom/top boundary), that will store a pointer to the connected adjacent `SWE_Block`.

The respective block approach can also be exploited for parallelisation: the different blocks would then be assigned to the available processors (or processor cores) – each processor (core) works on its share of blocks, while the program has to make sure to keep the values in the ghost cells up to date (which requires explicit communication in the case of distributed-memory computers).

The Time Step Loop

For each time step, our solver thus performs the following steps – each step is implemented via a separate member function of the class `SWE_Block`:

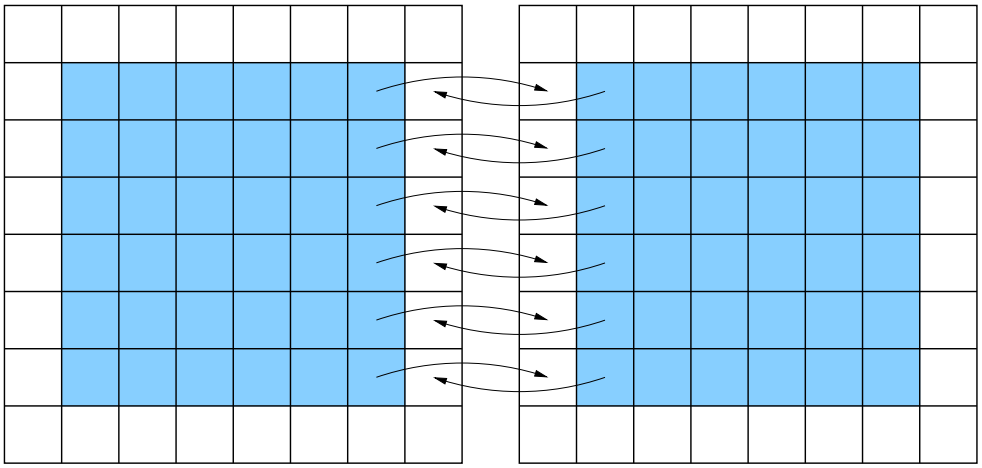

Figure 5: Exchange of values in ghost layers between two connected `SWE_Block`s.

1. set the values at the boundaries: `setBoundaryLayer()`;
2. compute the flux terms for all edges: `computeFluxes()`;
3. from the flux terms, compute the in/outflow balance for each cell, and compute the new values of the unknowns for the next time step: `eulerTimestep()`.

Visualisation with ParaView

The class `SWE_Block` contains further methods that will write the numerical solution into a sequence of files that can be read by the visualisation package ParaView (just enter the respective folder from ParaView – the files will be recognised and displayed as one project). ParaView allows to visualise the computed time-dependent solution (as "movie" or in single-step mode). ParaView is pretty self-explanatory for our purposes, but provides an online help for further instructions.

Visualisation "on the fly"

We also provide a CUDA implementation of the simulation code (requires a computer with a CUDA-capable GPU, together with the respective drivers – visit NVIDIA's website on CUDA for details on implementation). Apart from the fact that the simulation usually runs a lot faster on the GPU, the program is also capable of plotting the computing solution (water surface) "on the fly".