SWE –
Anatomy of a Parallel Shallow Water Code

CSCS-FoMICS-USI Summer School on Computer Simulations in Science and Engineering

Michael Bader
July 8–19, 2013
Teaching Parallel Programming Models . . .

Starting Point: Lecture on Parallel Programming

- classical approaches for shared & distributed memory: OpenMP and MPI
- “something more fancy” → GPU computing (CUDA, e.g.)
- motivating example to teach different models and compare their properties

“Motivating Example”:

- not just Jacobi or Gauß-Seidel
- not the heat equation again . . .
- inspired by a CFD code: “Nast” by Griebel et al.
- turned out to become shallow water equations
- and is heavily used for summer schools . . .
Towards Tsunami Simulation with SWE

Shallow Water Code – Summary

- Finite Volume discretization on regular Cartesian grids → simple numerics (but can be extended to state-of-the-art)
- patch-based approach with ghost cells for communication → wide-spread design pattern for parallelization
Towards Tsunami Simulation with SWE (2)

Shallow Water Code – Bells & Whistles

- included augmented Riemann solvers (D. George, R. LeVeque)
  → allows to simulate inundation
- developed towards hybrid parallel architectures
  → now runs on GPU clusters
Part I

Model and Discretization
Model: The Shallow Water Equations

Simplified setting (no friction, no viscosity, no Coriolis forces, etc.):

\[
\frac{\partial}{\partial t} \begin{bmatrix} h \\ hu \\ hv \end{bmatrix} + \frac{\partial}{\partial x} \begin{bmatrix} hu \\ hu^2 + \frac{1}{2} gh^2 \\ huv \end{bmatrix} + \frac{\partial}{\partial y} \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2} gh^2 \end{bmatrix} = \begin{bmatrix} 0 \\ -\frac{\partial}{\partial x}(ghb) \\ -\frac{\partial}{\partial y}(ghb) \end{bmatrix}
\]

Quantities and unknowns:

\( b \geq 0 \)

\( hu(x, t) \)

\( b(x) \)

\( h(x, t) \)
Model: The Shallow Water Equations

Simplified setting (no friction, no viscosity, no Coriolis forces, etc.):

\[
\begin{align*}
\frac{\partial}{\partial t} & \begin{bmatrix} h \\ hu \\ hv \end{bmatrix} + \frac{\partial}{\partial x} \begin{bmatrix} hu \\ hu^2 + \frac{1}{2} gh^2 \\ huv \end{bmatrix} + \frac{\partial}{\partial y} \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2} gh^2 \end{bmatrix} = \\
& \begin{bmatrix} 0 \\ -\frac{\partial}{\partial x} (ghb) \\ -\frac{\partial}{\partial y} (ghb) \end{bmatrix}
\end{align*}
\]

Write as generalized hyperbolic PDE:

- 2D setting, three quantities: \( q = (q_1, q_2, q_3)^T = (h, hu, hv)^T \)

\[
\begin{align*}
\frac{\partial}{\partial t} q + \frac{\partial}{\partial x} F(q) + \frac{\partial}{\partial y} G(q) &= S(q, x, y, t) \\
F(q) &= \begin{bmatrix} q_2 \\ \frac{q_2}{q_1}^2 + \frac{1}{2} gq_1^2 \\ \frac{q_2 q_3}{q_1} \end{bmatrix} \\
G(q) &= \begin{bmatrix} q_3 \\ \frac{q_2 q_3}{q_1} \end{bmatrix}
\end{align*}
\]
Model: The Shallow Water Equations

Simplified setting (no friction, no viscosity, no Coriolis forces, etc.):

\[
\begin{align*}
\frac{\partial}{\partial t} & \begin{bmatrix} h \\ hu \\ hv \end{bmatrix} + \frac{\partial}{\partial x} \begin{bmatrix} hu^2 + \frac{1}{2} gh^2 \\ huv \\ hv^2 + \frac{1}{2} gh^2 \end{bmatrix} + \frac{\partial}{\partial y} \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2} gh^2 \end{bmatrix} = \begin{bmatrix} 0 \\ -\frac{\partial}{\partial x} (ghb) \\ -\frac{\partial}{\partial y} (ghb) \end{bmatrix}
\end{align*}
\]

Derived from conservation laws:

- \( h \) equation: conservation of mass
- equations for \( hu \) and \( hv \): conservation of momentum
- \( \frac{1}{2} gh^2 \): averaged hydrostatic pressure due to water column \( h \), similar: bathymetry terms \( -\frac{\partial}{\partial x} (ghb) \) and \( -\frac{\partial}{\partial y} (ghb) \)
- may also be derived by vertical averaging from the 3D incompressible Navier-Stokes equations
Model: The Shallow Water Equations

Simplified setting (no friction, no viscosity, no Coriolis forces, etc.):

\[
\frac{\partial}{\partial t} \begin{bmatrix} h \\ hu \\ hv \end{bmatrix} + \frac{\partial}{\partial x} \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix} + \frac{\partial}{\partial y} \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix} = \begin{bmatrix} 0 \\ -\frac{\partial}{\partial x} (ghb) \\ -\frac{\partial}{\partial y} (ghb) \end{bmatrix}
\]

The ocean as “shallow water”??

- compare horizontal (\(\sim 1000\) km) to vertical (\(\sim 4\) km) length scale
- wave lengths large compared to water depth
- vertical flow may be neglected; movement of the “entire water column”
Finite Volume Discretisation

- discretise system of PDEs
  \[
  \frac{\partial}{\partial t} q + \frac{\partial}{\partial x} F(q) + \frac{\partial}{\partial y} G(q) = S(t, x, y)
  \]

- results from integral equation:
  \[
  \frac{\partial}{\partial t} \int_{t_n}^{t_{n+1}} \int_{\Omega} q \, d\omega \, dt + \int_{t_n}^{t_{n+1}} \int_{\partial \Omega} F(q) \cdot \vec{n} \, ds \, dt = \ldots
  \]

- use averaged quantities \( Q_{i,j}^{(n)} \) in finite volume elements \( \Omega_{ij} \):
  \[
  Q_{ij}(t) := \frac{1}{|\Omega_{ij}|} \int_{\Omega_{ij}} q \, d\omega \quad \Rightarrow \quad \frac{\partial}{\partial t} \int_{t_n}^{t_{n+1}} \int_{\Omega} q \, d\omega \, dt = |\Omega_{ij}| \left( Q_{i,j}^{(n+1)} - Q_{i,j}^{(n)} \right)
  \]

- What about the flux integral?
Finite Volume Discretisation (2)

- flux integral on Cartesian grids:

\[
\int_{t_n}^{t_{n+1}} \int_{\partial \Omega} \vec{F}(q) \cdot \vec{n} \, ds \, dt = \int_{t_n}^{t_{n+1}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} F(q(x_{i+\frac{1}{2}}, y, t)) - F(q(x_{i-\frac{1}{2}}, y, t)) \, dy \, dt \\
+ \int_{t_n}^{t_{n+1}} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} G(q(x, y_{j+\frac{1}{2}}, t)) - G(q(x, y_{i-\frac{1}{2}}, t)) \, dy \, dt
\]

- leads to explicit time stepping scheme:

\[
Q_{i,j}^{(n+1)} - Q_{i,j}^{(n)} = \frac{\Delta t}{\Delta y} \left( F(q(x_{i+\frac{1}{2}}, y, t_n)) - F(q(x_{i-\frac{1}{2}}, y, t_n)) \right) \\
+ \frac{\Delta t}{\Delta x} \left( G(q(x, y_{j+\frac{1}{2}}, t_n)) - G(q(x, y_{i-\frac{1}{2}}, t_n)) \right)
\]

- how to compute \(F_{i+\frac{1}{2},j}^{(n)} := F(q(x_{i+\frac{1}{2}}, y, t_n))\)?
Central and Upwind Fluxes

- define fluxes $F^{(n)}_{i+\frac{1}{2},j}, G^{(n)}_{i,j+\frac{1}{2}}, \ldots$ via 1D numerical flux function $\mathcal{F}$:

  $$
  F^{(n)}_{i+\frac{1}{2}} = \mathcal{F}(Q^{(n)}_i, Q^{(n)}_{i+1}) \quad G^{(n)}_{j-\frac{1}{2}} = \mathcal{F}(Q^{(n)}_{j-1}, Q^{(n)}_j)
  $$

- central flux:

  $$
  F^{(n)}_{i+\frac{1}{2}} = \mathcal{F}(Q^{(n)}_i, Q^{(n)}_{i+1}) := \frac{1}{2} \left( F(Q^{(n)}_i) + F(Q^{(n)}_{i+1}) \right)
  $$

  leads to unstable methods for convective transport

- upwind flux (here, for $h$-equation, $F(h) = hu$):

  $$
  F^{(n)}_{i+\frac{1}{2}} = \mathcal{F}(h^{(n)}_i, h^{(n)}_{i+1}) := \begin{cases} 
  hu_i & \text{if } u_{i+\frac{1}{2}} > 0 \\
  hu_{i+1} & \text{if } u_{i+\frac{1}{2}} < 0
  \end{cases}
  $$

  stable, but includes artificial diffusion
(Local) Lax-Friedrichs Flux

- classical **Lax-Friedrichs method** uses as numerical flux:
  \[
  F_{i+\frac{1}{2}}^{(n)} = \mathcal{F}(Q_i^{(n)}, Q_{i+1}^{(n)}) := \frac{1}{2} \left( F(Q_i^{(n)}) + F(Q_{i+1}^{(n)}) \right) - \frac{h}{2\tau} (Q_{i+1}^{(n)} - Q_i^{(n)})
  \]

- can be interpreted as central flux plus diffusion flux:
  \[
  \frac{h}{2\tau} (Q_{i+1}^{(n)} - Q_i^{(n)}) = \frac{h^2}{2\tau} \cdot \frac{Q_{i+1}^{(n)} - Q_i^{(n)}}{h}
  \]
  with diffusion coefficient \( \frac{h^2}{2\tau} \), where \( c := \frac{h}{\tau} \) is a velocity
  ("one grid cell per time step" → cmp. CFL condition)

- idea of **local Lax-Friedrichs** method: use the actual wave speed
  \[
  F_{i+\frac{1}{2}}^{(n)} := \frac{1}{2} \left( F(Q_i^{(n)}) + F(Q_{i+1}^{(n)}) \right) - \frac{a_{i+\frac{1}{2}}}{2} (Q_{i+1}^{(n)} - Q_i^{(n)})
  \]
Riemann Problems

- solve **Riemann problem** to obtain solution \( q(x_{i+\frac12}, y, t_n) \), etc.:  
- 1D treatment: solve shallow water equations with initial conditions  
  \[
  q(x_{i-\frac12}, t_n) = \begin{cases} 
    q_l = Q_{i-1}^{(n)} & \text{if } x < x_{i-\frac12} \\
    q_r = Q_i^{(n)} & \text{if } x > x_{i-\frac12}
  \end{cases}
  \]
- solution: two (left or right) outgoing waves (shock or rarefaction)
Riemann Problems

- solve **Riemann problem** to obtain solution \( q(x_{i+\frac{1}{2}}, y, t_n) \), etc.:
- 1D treatment: solve shallow water equations with initial conditions
  \[
  q(x_{i-\frac{1}{2}}, t_n) = \begin{cases} 
    q_l = Q_{i-1}^{(n)} & \text{if } x < x_{i-\frac{1}{2}} \\
    q_r = Q_{i}^{(n)} & \text{if } x > x_{i-\frac{1}{2}}
  \end{cases}
  \]
- solution: two (left or right) outgoing waves (shock or rarefaction)

![Diagram of Riemann problem](image)
Riemann Problems (2)

- wave propagation approach: split the jump into fluxes

\[ F(Q_i) - F(Q_{i-1}) - \Delta x \psi_{i-\frac{1}{2}} = \sum_p \alpha_p r_p \equiv \sum_p Z_p \quad \alpha_p \in \mathbb{R}. \]

- \( r_p \) the eigenvector of the linearised problem,
- \( \psi_{i-\frac{1}{2}} \) a fix for the source term (bathymetry)

- implementation will compute net updates:

\[ \mathcal{A}^+ \Delta Q_{i-1/2,j} = \sum_{p: \lambda_p > 0} Z_p \quad \mathcal{A}^- \Delta Q_{i-1/2,j} = \sum_{p: \lambda_p < 0} Z_p \]
The F-Wave Solver

- use Roe eigenvalues $\lambda_{1/2}^{\text{Roe}}$ to approximate the wave speeds:
  \[
  \lambda_{1/2}^{\text{Roe}}(q_l, q_r) = u^{\text{Roe}}(q_l, q_r) \pm \sqrt{g h^{\text{Roe}}(q_l, q_r)}
  \]
- with $h^{\text{Roe}}(q_l, q_r) = \frac{1}{2}(h_l + h_r)$ and $u^{\text{Roe}}(q_l, q_r) = \frac{u_l \sqrt{h_l} + u_r \sqrt{h_r}}{\sqrt{h_l} + \sqrt{h_r}}$
- eigenvectors $r_{1/2}^{\text{Roe}}$ for wave decomposition defined as
  \[
  r_1^{\text{Roe}} = \begin{pmatrix} 1 \\ \lambda_1^{\text{Roe}} \end{pmatrix} \quad r_2^{\text{Roe}} = \begin{pmatrix} 1 \\ \lambda_2^{\text{Roe}} \end{pmatrix}
  \]
- leads to net updates (source terms still missing):
  \[
  A^- \Delta Q := \sum_{p: \{\lambda_p^{\text{Roe}} < 0\}} \alpha_p r_p \quad A^+ \Delta Q := \sum_{p: \{\lambda_p^{\text{Roe}} > 0\}} \alpha_p r_p
  \]
- with $\alpha_{1/2}$ computed from
  \[
  \begin{pmatrix} 1 \\ \lambda_1^{\text{Roe}} \\ \lambda_2^{\text{Roe}} \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = F(Q_i) - F(Q_{i-1})
  \]
Finite Volume on Cartesian Grids

Unkowns and Numerical Fluxes:

- (averaged) unknowns $h$, $hu$, $hv$, and $b$ located in cell centers
- two sets of “net updates” or “numerical fluxes” per edge; here: $A^+ \Delta Q_{i-1/2,j}$, $B^- \Delta Q_{i,j+1/2}$ (“wave propagation form”)
Flux Form vs. Wave Propagation Form

- numerical scheme in flux form:

\[ Q_{i,j}^{(n+1)} = Q_{i,j}^{(n)} - \frac{\Delta t}{\Delta x} \left( F_{i+\frac{1}{2},j}^{(n)} - F_{i-\frac{1}{2},j}^{(n)} \right) - \frac{\Delta t}{\Delta y} \left( G_{i,j+\frac{1}{2}}^{(n)} - G_{i,j-\frac{1}{2}}^{(n)} \right) \]

where \( F_{i+\frac{1}{2},j}^{(n)}, G_{i,j+\frac{1}{2}}^{(n)}, \ldots \) approximate the flux functions \( F(q) \) and \( G(q) \) at the grid cell boundaries.

- Wave propagation form:

\[ Q_{i,j}^{n+1} = Q_{i,j}^{n} - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-1/2,j}^n + A^- \Delta Q_{i+1/2,j}^n \right) \]
\[ - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-1/2}^n + B^- \Delta Q_{i,j+1/2}^n \right). \]

where \( A^+ \Delta Q_{i-1/2,j}, B^- \Delta Q_{i,j+1/2}^n, \) etc. are net updates.

- difference in implementation: compute one “flux term” or two “net updates” for each edge.
Time Stepping: Splitting or Not?

- With **Dimensional Splitting**:

\[
Q_{i,j}^* = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-1/2,j}^n + A^- \Delta Q_{i+1/2,j}^n \right)
\]

\[
Q_{i,j}^{n+1} = Q_{i,j}^* - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-1/2}^* + B^- \Delta Q_{i,j+1/2}^* \right).
\]

two sequential “sweeps” of Riemann solves on horizontal vs. vertical edges

- vs. “un-split” method: (**currently used in SWE**)

\[
Q_{i,j}^{n+1} = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-1/2,j}^n + A^- \Delta Q_{i+1/2,j}^n \right)
\]

\[
- \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-1/2}^n + B^- \Delta Q_{i,j+1/2}^n \right).
\]
allows to combine loops on horizontal and vertical edges
Time Stepping

CFL Condition:

- we only consider neighbour cells for a time step ⇒ information must not travel faster than one cell per timestep!
- thus: timesteps need to consider characteristic wave speeds
- rule of thumb: wave speed depends on water depth, $\lambda = \sqrt{gh}$
- in SWE: Riemann solvers will compute local wave speeds ⇒ maximum-reduction necessary to find global time step

Adaptive time step control forces sequential main loop:

1. solve Riemann problems, compute wave speeds
2. compute maximum wave speed and infer global $\Delta t$
3. update unknowns
References & Literature

- George: Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation, Journal of Computational Physics 227 (6), 2008
Part II

Parallel Programming Patterns

Reference: Mattson, Sanders, Massingill, Patterns for Parallel Programming. Addison-Wesley, 2005.
Finding Concurrency

Common rule:

*Before you start parallelising your code, make sure the serial version is perfectly optimised!*

Pro:

- parallelising a badly optimised serial algorithm leads to a badly optimised parallel algorithm
- **use an asymptotically optimal algorithm!**
  for large problems (that are worth being parallelised) asymptotics is crucial

Contra:

- **exploit all available concurrency in your problem**
  (your optimised serial code might have unnecessary sequential parts)
- the fastest serial algorithm is not necessarily the fastest parallel algorithm
Finding Concurrency – Task Decomposition

Decompose your problem into tasks that can execute concurrently!

Consider “un-split” time stepping:

\[
\forall i, j : Q_{i,j}^{n+1} = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-1/2,j}^n + A^- \Delta Q_{i+1/2,j}^n \right) \\
- \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-1/2}^n + B^- \Delta Q_{i,j+1/2}^n \right)
\]

Concurrent tasks:

1. compute net updates (i.e., solve Riemann problems)
   \( A^+ \Delta Q_{i-1/2,j}^n, B^+ \Delta Q_{i,j-1/2}^n \) for all (vertical and horizontal) edges

2. update quantities \( Q_{i,j}^{n+1} \) in all cells

or: for all cells, compute net updates (on local edges) and update quantities \( Q_{i,j}^{n+1} \) (requires two arrays for \( Q_{i,j}^n \) and \( Q_{i,j}^{n+1} \), resp.)
Finding Concurrency – Task Decomposition

Decompose your problem into tasks that can execute concurrently!

Consider Dimensional Splitting:

\[
Q^n_{i,j} = Q^n_{i,j} - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q^n_{i-1/2,j} + A^- \Delta Q^n_{i+1/2,j} \right)
\]

\[
Q^{n+1}_{i,j} = Q^*_{i,j} - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q^*_{i,j-1/2} + B^- \Delta Q^*_{i,j+1/2} \right).
\]

Concurrent tasks:

1. compute net updates on all vertical edges \((A^+ \Delta Q^n_{i-1/2,j}, \text{ etc.})\)
   1a. update intermediate quantities \(Q^*_{i,j}\) in all cells

2. compute net updates on all horizontal edges \((B^+ \Delta Q^n_{i,j-1/2}, \text{ etc.})\)
   2a. update quantities \(Q^{n+1}_{i,j}\) in all cells
Finding Concurrency – Data Decomposition

Decompose your data into units that can operated on relatively independently!

Consider Dimensional Splitting:

\[ Q_{i,j}^* = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-1/2,j}^n + A^- \Delta Q_{i+1/2,j}^n \right) \]

\[ Q_{i,j}^{n+1} = Q_{i,j}^* - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-1/2}^* + B^- \Delta Q_{i,j+1/2}^* \right). \]

Data Decomposition:

1. computation of \( Q_{i,j}^* \): distribute data row-wise, as computation is independent for different \( j \)
2. update of \( Q_{i,j}^{n+1} \): distribute data column-wise, as computation is independent for different \( i \)
Finding Concurrency – Data Decomposition

Decompose your data into units that can operated on relatively independently!

Consider “un-split” time stepping:

\[ \forall i, j: \, Q_{i,j}^{n+1} = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-1/2,j}^n + A^- \Delta Q_{i+1/2,j}^n \right) \]

\[ - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-1/2}^n + B^- \Delta Q_{i,j+1/2}^n \right) \]

Concurrent tasks:

- compute net updates requires left/right and top/down neighbours ⇒ no “perfect” data decomposition possible
- partitioning of data will require extra care at boundaries of the partitions
- and: (seemingly trivial) do not decompose quantities in \( Q_{i,j} \)
Task and Data Decomposition – “Forces”

Flexibility:
- be flexible enough to adapt to different implementation requirements
- for example: do not concentrate on a single parallel platform or programming model

Efficiency:
- solution needs to scale efficiently with the size of the computer
- task and data decomposition need to provide enough tasks to keep all processing elements busy

Simplicity:
- complex enough to solve the task, but simple enough to keep program maintainable
Identifying Dependencies Between Tasks

Group Tasks:

*Group your tasks to simplify the managing of dependencies*

Order Tasks:

*Given a collection of tasks into logically related groups, order these task groups to satisfy constraints*

Data Sharing:

*Given a data and task decomposition, how is data shared among the tasks?*
Element Updates as Task Groups

Consider “un-split” time stepping:

\[ \forall i, j : Q_{i,j}^{n+1} = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-1/2,j}^n + A^- \Delta Q_{i+1/2,j}^n \right) \]
\[ - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-1/2}^n + B^- \Delta Q_{i,j+1/2}^n \right) \]

Grouped Tasks:

- solve Riemann problems on the four cell edges
- update quantities \( Q_{i,j} \) from the net updates

Data Dependencies:

- tasks access quantities \( Q_{i\pm1,j\pm1}^n \) of neighbour cells
  \[ \Rightarrow \] two copies required for \( Q_{i,j}^n \) and \( Q_{i,j}^{n+1} \)
- Riemann problem computed twice for each edge?
Riemann Solves and Updates as Task Groups

Consider Dimensional Splitting:

\[ Q_{i,j}^* = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-1/2,j}^n + A^- \Delta Q_{i+1/2,j}^n \right) \]

\[ Q_{i,j}^{n+1} = Q_{i,j}^* - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-1/2}^* + B^- \Delta Q_{i,j+1/2}^* \right). \]

Separate Task Groups (for each of the two steps):

- solve Riemann problems on all horizontal (vertical) cell edges
- update quantities \( Q_{i,j} \) of an entire column (row)

Data Dependencies:

- tasks access neighbours in either row or column direction
- requires extra storage to compute the net updates (results of the Riemann problems)
Consider “un-split” time stepping:

\[
\forall i, j : \quad Q_{i,j}^{n+1} = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-1/2,j}^n + A^- \Delta Q_{i+1/2,j}^n \right)
- \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-1/2}^n + B^- \Delta Q_{i,j+1/2}^n \right)
\]

where \( \Delta t \) results from wave propagation speeds

Sequential Order of Tasks:

1. solve Riemann problems on the four cell edges
   (compute wave propagation speeds as partial results)
2. determine maximum wave speed for CFL condition \( \sim \Delta t \)
3. update quantities \( Q_{i,j} \) from the net updates
The Geometric Decomposition Pattern

How can your algorithm be organized around a data structure that has been decomposed into concurrently updatable “chunks”?

Partitioning (how to select your “chunks”):

- w.r.t. size, shape, etc. (“granularity” of parallelism)
- multiple levels of partitioning necessary?

Organization of parallel updates:

- need to access water height, momentum components and bathymetry from neighbour cells (possible in other partition)
- need to access net updates from neighbour partition? (alternative: compute on all involved partitions?)
1D Domain Decomposition – Slice-Oriented

Discussion:

- degenerates for large number of partitions: thin slices, lots of data exchange required at (long!) boundaries
- for dimensional splitting: slices match dependencies (vertical or horizontal) but alternating slices required for the two update steps
Discussion:

- length of domain boundaries (communication volume)
- fit arbitrary number of partitions to layout of boxes
3D Domain Decomposition – Cuboid-Oriented
“Patches” Concept for Domain Decomposition

Discussion:

- more fine-grain load distribution
- “empty patches” improve representation of complicated domains
  - overhead for additional, interior boundaries
  - requires scheme to assign patches to processes
Part III

SWE Software Design
Basic Structure: Cartesian Grid Block

Spatial Discretization:

- regular Cartesian meshes; later: allow multiple patches
- ghost layers to implement boundary conditions; connect multiple patches (complicated domains, parallelization)
Basic Structure: Cartesian Grid Block

Data Structure:

- arrays h, hu, hv, and b to hold water height, momentum components and bathymetry data
- “column major” layout: j the “faster running” index in h[i][j]
Main Loop – Euler Time-stepping

\[\textbf{while}( \ t < \ ... \ ) \ \{}\]
\[
  \quad // \textit{set boundary conditions} \\
  \quad \text{splash.setGhostLayer();} \\
\]
\[
  \quad // \textit{compute fluxes on each edge} \\
  \quad \text{splash.computeNumericalFluxes();} \\
\]
\[
  \quad // \textit{set largest allowed time step:} \\
  \quad \text{dt} = \text{splash.getMaxTimestep();} \\
  \quad \text{t} \text{ += dt;} \\
\]
\[
  \quad // \textit{update unknowns in each cell} \\
  \quad \text{splash.updateUnknowns(dt);} \\
\]\n
→ defines interface for abstract class SWE\_Block
Set Ghost Layers – Boundary Conditions

Split into two methods:

- `setGhostLayer()`: interface function in SWE_Block, needs to be called by main loop
- `setBoundaryConditions()`: called by `setGhostLayer()`; sets “real” boundary conditions (WALL, OUTFLOW, etc.)

```cpp
switch(boundary[BND_LEFT]) {
    case WALL:
    {
        for(int j=1; j<=ny; j++) {
            h[0][j] = h[1][j];    b[0][j] = b[1][j];
            hu[0][j] = -hu[1][j]; hv[0][j] = hv[1][j];
        }
        break;
    }
    case OUTFLOW:
    { /* ... */
```

`Michael Bader: SWE – Anatomy of a Parallel Shallow Water Code`

*Computer Simulations in Science and Engineering, July 8–19, 2013*
Compute Numerical Fluxes

main loop to compute net updates on left/right edges:

```cpp
for(int i=1; i < nx+2; i++) {
    for(int j=1; j < ny+1; j++) {
        float maxEdgeSpeed;
        wavePropagationSolver.computeNetUpdates(
            h[i-1][j], h[i][j],
            hu[i-1][j], hu[i][j],
            b[i-1][j], b[i][j],
            hNetUpdatesLeft[i-1][j-1], hNetUpdatesRight[i-1][j-1],
            huNetUpdatesLeft[i-1][j-1], huNetUpdatesRight[i-1][j-1],
            maxEdgeSpeed
        );
        maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);
    }
}
```

(cmp. file SWE_WavePropagationBlock.cpp)
Compute Numerical Fluxes (2)

main loop to compute net updates on top/bottom edges:

```cpp
for (int i = 1; i < nx+1; i++) {
    for (int j = 1; j < ny+2; j++) {
        float maxEdgeSpeed;
        wavePropagationSolver.computeNetUpdates(
            h[i][j-1], h[i][j],
            hv[i][j-1], hv[i][j],
            b[i][j-1], b[i][j],
            hNetUpdatesBelow[i-1][j-1], hNetUpdatesAbove[i-1][j-1],
            hvNetUpdatesBelow[i-1][j-1], hvNetUpdatesAbove[i-1][j-1],
            maxEdgeSpeed
        );
        maxWaveSpeed = std::max(maxWaveSpeed, maxEdgeSpeed);
    }
}
```

(cmp. file SWE_WavePropagationBlock.cpp)
Determine Maximum Time Step

- variable maxWaveSpeed holds maximum wave speed
- updated during computation of numerical fluxes in method computeNumericalFluxes():
  \[ \text{maxTimestep} = \text{std}::\text{min}( \text{dx}/\text{maxWaveSpeed}, \text{dy}/\text{maxWaveSpeed} ) \];
- simple “getter” method defined in class SWE Block:
  ```cpp
  float getMaxTimestep() { return maxTimestep; }
  ```
- hence: getMaxTimestep() for current time step should be called after computeNumericalFluxes()
- in general: in many situations, the maximum computation inhibits certain optimizations → fixed time step probably faster!
Technische Universität München

Update Unknowns – Euler Time Stepping

for(int i=1; i < nx+1; i++) {
    for(int j=1; j < ny+1; j++) {
        h[i][j] -= dt/dx * (hNetUpdatesRight[i-1][j-1]
                        + hNetUpdatesLeft[i][j-1])
                 + dt/dy * (hNetUpdatesAbove[i-1][j-1]
                        + hNetUpdatesBelow[i-1][j]);

        hu[i][j] -= dt/dx * (huNetUpdatesRight[i-1][j-1]
                          + huNetUpdatesLeft[i][j-1]);

        hv[i][j] -= dt/dy * (hvNetUpdatesAbove[i-1][j-1]
                          + hvNetUpdatesBelow[i-1][j]);

        /* ... */
    }
}

(cmp. file SWE_WavePropagationBlock.cpp)
Goals for (Parallel) Implementation

Spatial Discretization:
- allow different parallel programming models
- and also to switch between different numerical models
⇒ class hierarchy of numerical vs. programming models

Hybrid Parallelization:
- support two levels of parallelization (such as shared/distributed memory, CPU/GPU, etc.)
- coarse-grain parallelism across Cartesian grid patches
- fine-grain parallelism on patch-local operations
⇒ separate fine-grain and coarse-grain parallelism (plug&play principle)
abstract class SWE_Block:

- base class to hold data structures (arrays h, hu, hv, b)
- manipulates ghost layers
- methods for initialization, writing output, etc.
- defines interface for main time-stepping loop: computeNumericalFluxes(), updateUnknowns(), ...
derived classes:

- for different model variants: SWE_RusanovBlock, SWE_WavePropagationBlock, ...
- for different programming models: SWE_BlockCUDA, SWE_BlockArBB, ...
- override computeNumericalFluxes(), updateUnknowns(), ... → methods relevant for parallelization
Example: `SWE_WavePropagationBlockCuda`

```cpp
class SWE_WavePropagationBlockCuda: public SWE_BlockCUDA {
    /*-- definition of member variables skipped --*/
    public:
        // compute a single time step (net-updates + update of the cells).
        void simulateTimestep(float i_dT);
        // simulate multiple time steps (start and end time as parameters)
        float simulate(float, float);
        // compute the numerical fluxes (net-update formulation here).
        void computeNumericalFluxes();
        // compute the new cell values.
        void updateUnknowns(const float i_deltaT);
};
```

(in file `SWE_WavePropagationBlockCuda.hh`)
Part IV

SWE Parallelisation
Patches of Cartesian Grid Blocks

Spatial Discretization:

- regular Cartesian meshes; allow multiple patches
- ghost and copy layers to implement boundary conditions, for more complicated domains, and for parallelization
Loop-Based Parallelism within Patches

Computing the Net Updates

- compute net updates on left/right edges:
  
  ```
  for(int i=1; i < nx+2; i++) in parallel {
    for(int j=1; j < ny+1; j++) in parallel {
      float maxEdgeSpeed;
      fWaveComputeNetUpdates(9.81, h[i-1][j], h[i][j], hu[i-1][j], hu[i][j], /* ... */);
    }
  }
  ```

- compute net updates on top/bottom edges:
  
  ```
  for(int i=1; i < nx+1; i++) in parallel {
    for(int j=1; j < ny+2; j++) in parallel {
      fWaveComputeNetUpdates(9.81, h[i][j-1], h[i][j], hv[i][j-1], hv[i][j], /* ... */);
    }
  }
  ```

(function fWaveComputeNetUpdates() defined in file solver/FWaveCuda.h)
Computing the Net Updates

Options for Parallelism

Parallelization of computations:

- compute all vertical edges in parallel
- compute all horizontal edges in parallel
- compute vertical & horizontal edges in parallel (task parallelism)

Parallel access to memory:

- concurrent read to variables h, hu, hv
- exclusive write access to net-update variables on edges
Loop-Based Parallelism within Patches (2)

Updating the Unknowns

- update unknowns from net updates on edges:

```c
for(int i=1; i < nx+1; i++) in parallel {
    for(int j=1; j < ny+1; j++) in parallel {
        h[i][j] -= dt/dx * (hNetUpdatesRight[i-1][j-1]
                         + hNetUpdatesLeft[i][j-1])
                         + dt/dy * (hNetUpdatesAbove[i-1][j-1]
                                  + hNetUpdatesBelow[i-1][j]);
        hu[i][j] -= dt/dx * (huNetUpdatesRight[i-1][j-1]
                             + huNetUpdatesLeft[i][j-1]);
    }
}
```

...
Updating the Unknowns
Options for Parallelism

Parallelization of computations:

- compute all cells in parallel

Parallel access to memory:

- concurrent read to net-updates on edges
- exclusive write access to variables \( h, hu, hv \)

“Vectorization property”:

- exactly the same code for all cell!
Exchange of Values in Ghost/Copy Layers

Straightforward Approach:

- boundary conditions OUTFLOW, WALL vs. CONNECT or PARALLEL
- disadvantage: method setGhostLayer() needs to be implemented for each derived class
Exchange of Values in Ghost/Copy Layers (2)

Implemented via Proxy Objects:

- `grabGhostLayer()` to write into ghost layer
- `registerCopyLayer()` to read from copy layer
- both methods return a proxy object (class `SWE_Block1D`) that references one row/column of the grid
Direct-Neighbour vs. “Diagonal” Communication

2-step scheme to exchange data of “diagonal” ghost cells:

- several “hops” replace diagonal communication
- slight increase of volume of communication (bandwidth), but reduces number of messages (latency)
- similar in 3D (26 neighbours → 6 neighbours!)
MPI Parallelization
– Exchange of Ghost/Copy Layers

SWE_Block1D* leftInflow = splash.grabGhostLayer(BND_LEFT);
SWE_Block1D* leftOutflow = splash.registerCopyLayer(BND_LEFT);

SWE_Block1D* rightInflow = splash.grabGhostLayer(BND_RIGHT);
SWE_Block1D* rightOutflow = splash.registerCopyLayer(BND_RIGHT);

MPI_Sendrecv(leftOutflow->h.elemVector(), 1, MPI_COL, leftRank, 1,
             rightInflow->h.elemVector(), 1, MPI_COL, rightRank, 1,
             MPI_COMM_WORLD,&status);

MPI.Sendrecv(rightOutflow->h.elemVector(), 1, MPI_COL, rightRank,4,
             leftInflow->h.elemVector(), 1, MPI_COL, leftRank, 4,
             MPI_COMM_WORLD,&status);

(cmp. file examples/swe_mpi.cpp)
MPI – Some Speedups

- 1 MPI process per core
- (expensive) augmented Riemann solvers
Speedups for MPI/OpenMP

- 1 MPI process per node, 8 OpenMP threads (1 per core)
- straightforward OpenMP parallelization of for-loops
Speedups for MPI/OpenMP

- 1 MPI process per node, 8 OpenMP threads (1 per core)
- hybrid f-Wave/aug. Riemann solver → poor load balancing
Teaching Parallel Programming with SWE

SWE in Lectures, Tutorials, Lab Courses:

- non-trivial example, but model & implementation easy to grasp
- allows different parallel programming models (MPI, OpenMP, CUDA, Intel TBB/ArBB, OpenCL, . . .)
- prepared for hybrid parallelisation

Some Extensions:

- ASAGI - parallel server for geoinformation (S. Rettenberger, Master’s thesis)
- OpenGL real-time visualisation of results (T. Schnabel, student project; extended by S. Rettenberger)

→ [http://www5.in.tum.de/SWE/](http://www5.in.tum.de/SWE/)
→ [https://github.com/TUM-I5](https://github.com/TUM-I5)
Part V

Workshop – SWE Parallelisation
MPI Communication Between Patches

Extend sequential SWE program `swe_serial.cpp`:

- goal: one patch (SWE_Block per MPI process)
- establish assignment of patches to MPI ranks (“who is my neighbour?”)
- implement exchange between ghost & copy cells (preferably via proxy objects)
- parallelize adaptive time step control
- produce speed-up graphs (strong and weak scaling)

Possible extensions: (for the ambitious . . . )

- compare blocking vs. non-blocking communication
- try overlapping communication and computation
- allow multiple patches per MPI process
Loop Parallelism in SWE Using OpenMP

What should be done before starting with OpenMP?

- determine most time-consuming parts of your code (→ week 2)
- use option “guided auto-parallelism” of Intel compiler (→ welcome to try, but does not give many hints for SWE)

Extend MPI-parallel SWE program towards MPI+OpenMP:

- what are the most time-consuming loops in SWE?
- ToDo: loop parallelism for these loops using #pragma ...
- test performance of OpenMP vs. MPI implementation

Possible extensions: (for the ambitious . . .)

- parallelize adaptive time step control (reduction)
- multiple-patch version: try OpenMP on patches