HPSC

Result presentation

06.01.2025

Audric Demeure and Luca Santoro

Serial Code

Interpolation methods

Let the grid position given by: $i = \left\lfloor \frac{x}{\Delta x} \right\rfloor, \ j = \left\lfloor \frac{y}{\Delta y} \right\rfloor$.

Nearest Neighbor

Simply rounds the coordinates pair $\left(\frac{x}{\Delta x}, \frac{y}{\Delta y} \right)$ to the nearest integer index (i, j) and returns the data value at that single grid point:

$$Q(x, y) = Q_{i,j}$$

Bilinear interpolation

Using the interpolation points: $(x_1, y_1) = (i\Delta x, j\Delta y)$ and $(x_2, y_2) = ((i+1)\Delta x, (j+1)\Delta y)$, the weighted coefficients are: $w_x = \frac{x_2 - x}{\Delta x}, \ w_y = \frac{y_2 - y}{\Delta y}$. The interpolated value is then:

$$Q(x, y) = w_x w_y Q_{11} + (1 - w_x) w_y Q_{21} + w_x (1 - w_y) Q_{12} + (1 - w_x)(1 - w_y) Q_{22}$$

Note: $Q(x, y) = Q_{i,j}$ if (i, j) outside domain.

Analysis of η Evaluation

Initial η Evaluation

$$\eta_{i,j}^{n+1} = \eta_{i,j}^{n} - \Delta t \cdot h_{i,j}\left(\frac{u_{i+1,j} - u_{i,j}}{\Delta x} + \frac{v_{i,j+1} - v_{i,j}}{\Delta y}\right)$$

Corrected η Evaluation

$$\frac{\eta_{i,j}^{n+1} - \eta_{i,j}^{n}}{\Delta t} = -\frac{h\left(x_{u_{i+1,j}}\right)u_{i+1,j} - h\left(x_{u_{i,j}}\right)u_{i,j}}{\Delta x} - \frac{h\left(x_{v_{i,j+1}}\right)v_{i,j+1} - h\left(x_{v_{i,j}}\right)v_{i,j}}{\Delta y}$$

$$\Longleftrightarrow \eta_{i,j}^{n+1} = \eta_{i,j}^{n} - \Delta t\left(\frac{h\left(x_{u_{i+1,j}}\right)u_{i+1,j} - h\left(x_{u_{i,j}}\right)u_{i,j}}{\Delta x} + \frac{h\left(x_{v_{i,j+1}}\right)v_{i,j+1} - h\left(x_{v_{i,j}}\right)v_{i,j}}{\Delta y}\right)$$

Arithmetic Intensity

In the algorithm, the most computationally intensive routines occur in the double-nested loops of `update_eta` and `update_velocities` that loops over $i = 0, ..., n_x - 1$ and $j = 0, ..., n_y - 1$.

`update_eta`:

```
eta_ij = GET(eta, i, j)
        - c1_x * (h_ui_plus_1_j * u_ip1_j - h_ui_j * u_i_j)
        - c1_y * (h_vi_j_plus_1 * v_i_jp1 - h_vi_j * v_i_j);
```

Roughly 10 FLOPs per grid cell (i,j).

Arithmetic Intensity (ii)

`update_velocities`

```
u_ij = (1. - c2) * GET(u, i, j)
     - (c1 / param.dx) * (eta_ij - eta_imj);

v_ij = (1. - c2) * GET(v, i, j)
       - (c1 / param.dy) * (eta_ij - eta_ijm);
```

Roughly 8 FLOPs per cell.

Putting both together, we have:

$$\left(10_{\{\text{update\_eta}\}} + 8_{\{\text{update\_velocities}\}}\right) * \text{nx} * \text{ny} = 18 * (\text{nx} * \text{ny}) \text{ FLOPs per timestep.}$$

Over nt timesteps, the total is $\approx 18 * \text{nx} * \text{ny} * \text{nt}$ FLOPs.

Arithmetic Intensity (iii)

Memory Accesses per Kernel

- `update_eta` ~13 reads + 1 write ≈ 14 total memory operations
- `update_velocities` ~5 reads + 2 writes ≈ 7 total memory operations

So combined we have $7 + 14 \approx 21$ memory access. Then each double is 8 byte so we have $21 \text{ memory ops per cell} * 8 \frac{\text{byte}}{\text{op}} = 168 \text{ bytes}$.

Arithmetic Intensity With 18 FLOPs per cell and 168 bytes per cell:

$$\frac{18 \text{ FLOPs}}{168 \text{ bytes}} \approx 0.11 \text{ FLOPs/byte}$$

Such a low Arithmetic intensity ≈0.1 usually indicates **memory-bound behavior**.

Arithmetic Intensity (iv)

Comparing CPU vs. Memory Time

CPU-limited time: $T_{\text{CPU}} = \frac{\text{FLOPs}}{\text{Peak FLOP Rate}} = \frac{18 \times (\text{Nx} \times \text{Ny})}{2.8 \times 10^{12}\,\text{FLOPs/s}}$

For Nx = 100 and Ny = 100 and total $\text{FLOPs} \approx 1.8 \times 10^5$ per timestep.

Thus we have : $T_{\text{CPU}} \approx \frac{1.8 \times 10^5}{2.8 \times 10^{12}} = 6.4 \times 10^{-8}\,s \approx 0.06\,\mu\ \text{s per timestep.}$

Memory-limited time:

$$T_{\text{memory}} = \frac{\text{bytes accessed}}{\text{Memory Bandwidth}} = \frac{168 \times (\text{nx} \times \text{ny})}{200 \times 10^9\ \text{bytes/s}}$$

with nx=ny=100, total byte $\approx 168 * 10^4 = 1.68 * 10^4\ \text{bytes.}$

$T_{\text{memory}} \approx \frac{1.68 \times 10^6}{2 \times 10^{11}} = 8.4 \times 10^{-6}\,s = 8.4\,\mu\ \text{s per timestep.}$

$T_{\text{CPU}} \ll T_{\text{memory}}$ so we can conclude that the code is memory-bounded.

Note: $T(n) = \alpha + \beta n \approx \beta n$ since $\alpha \ll \beta n$ (for large data size) $\rightarrow$ focus on the bandwidth-limited part.

Memory Access Bottleneck

Problem: In the original code, loops were written with indices in the wrong order, causing non-contiguous memory access. This led to excessive cache misses and poor performance.

Solution: By reordering the loops to match the fastest-moving index in the innermost loop, the CPU can fetch data more efficiently from cache.

Indexing Order	**Speedup**
`i, j`	1.00
`j, i`	2,41

Keynote: Contiguous memory access leverages cache lines more effectively, drastically reducing memory latency and increasing overall speed.

Numerical Stability

Theoretical CFL Analysis

The CFL (Courant-Friedrichs-Lewy) condition applied to shallow water equations takes the following form:

$$\lambda \propto c\left(\frac{\Delta t}{\Delta x}\right) \leq 1$$

where:

- $c \approx \sqrt{gh}$ represents the characteristic velocity.

In two dimensions, this condition is modified to:

$$\lambda \approx \sqrt{2}c\left(\frac{\Delta t}{\Delta x}\right) \leq 1$$

For our analysis, we selected a maximum bathymetric value of $h_{\text{max}} \approx 21.875$ m, which allows us to express the timestep condition as:

$$\Delta t \leq \frac{\Delta x}{\sqrt{2}c} \approx 0.048\Delta x$$

Numerical Investigation

We conducted multiple simulations with varying Δt and Δx (resp. Δy) to experimentally determine the relationship between Δx and Δt:

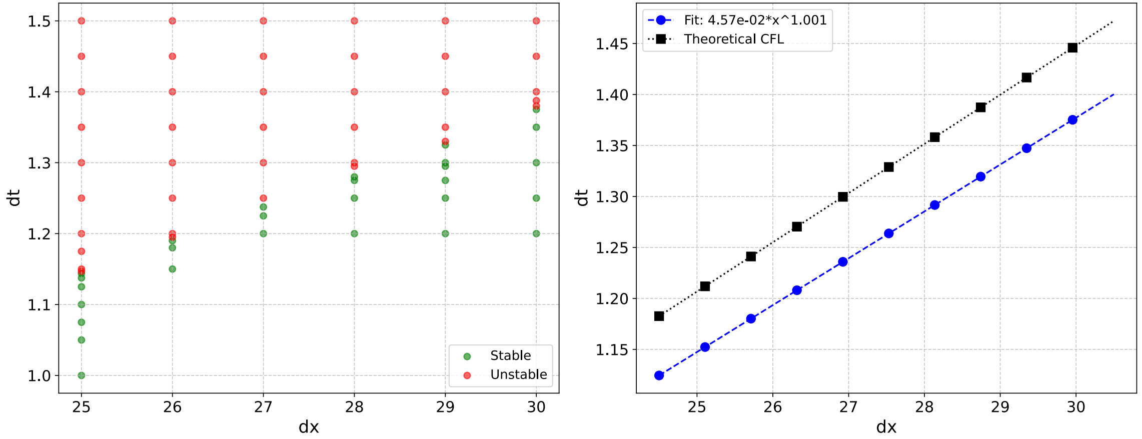


Figure 1: Numerical CFL condition determination.

Findings and Implementation

The results confirm that the CFL relationship is indeed linear between Δx and Δt. However, we observe a difference between the theoretical and numerical relationships, which might be partially explained by the approximation $h \to h_{\text{max}}$.

For the following analysis, we used the following parameters:

- Spatial steps: $\Delta x = \Delta y = 25$
- A more conservative CFL condition:

$$\lambda = c\left(\frac{\Delta t}{\Delta x}\right) \leq \frac{1}{2}$$

This conservative condition was proposed by A.I. Delis and Th. Katsaounis in their 2005 paper “Numerical solution of the two-dimensional shallow water equations by the application of relaxation methods”.

MPI Parallelization

Principle

The computational domain is divided into $P_x \times P_y$ partitions, where each MPI process operates on a distinct subdomain, reducing computation time by distributing the workload while minimizing inter-process communication through strategic splitting in both x and y directions.

Domain Decomposition: Cartesian Topology

- **MPI_Dims_create:** Determines a suitable 2D factorization of the total number of processes, giving us $\mathrm{dims}[0] = \mathrm{P}_\mathrm{x}$ and $\mathrm{dims}[1] = \mathrm{P}_\mathrm{y}$.
- **MPI_Cart_create:** Creates a 2D Cartesian communicator (`cart_comm`).
- **MPI_Cart_coords:** Retrieves each process's (x, y) coordinates in the 2D grid.
- **MPI_Cart_shift:** Identifies neighbors (`LEFT, RIGHT, UP, DOWN`) needed for sending/receiving ghost cells.

  These function were used to ***minimize inter-process communication.***
- **Our configuration:** Each MPI rank holds local arrays for η, u, v, and the interpolated bathymetry `h_interp`.

Local and Global Indices

- Each rank r computes the `start_i`, `start_j`, `end_i`, and `end_j` of its subdomain.
- The local arrays are sized:

$$\texttt{u} \to (\texttt{local\_nx} + 1) \times \texttt{local\_ny}, \quad \texttt{v} \to \texttt{local\_nx} \times (\texttt{local\_ny} + 1),$$

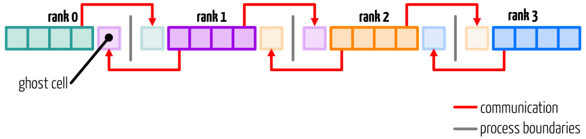
$$\texttt{h\_interp} \to \texttt{local\_nx} \times \texttt{local\_ny} \ .$$

- Conversion between *global* (i, j) and *local* $(i_{\text{rank}}, j_{\text{rank}})$ is done by:

$$i_{\text{rank}} = i - \texttt{start\_i}, \quad j_{\text{rank}} = j - \texttt{start\_j} \ .$$

Ghost Cells and Boundary Exchange

- To update η, u, and v near boundaries, we use neighboring data (*ghost cells*).



- **MPI_Isend / MPI_Irecv:** Non-blocking send/receive pairs exchange boundary rows or columns with adjacent ranks.
- After posting sends and receives, calls to **MPI_Waitall** ensure data arrival before the next computation step.
- This approach *hides* some communication time by overlapping with computations.

Updating η and u, v in Parallel

- Each rank performs the same numerical updates on η, u, and v, but restricted to its subdomain.
- **update_eta:**
  - Uses local arrays plus ghost cells from neighbors:

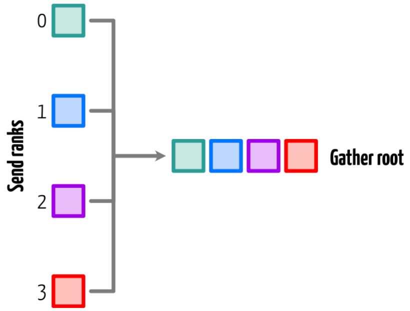
$$\eta_{i,j}^{n+1} = \eta_{i,j}^{n} - \Delta t \left[\frac{\partial (h\, u)}{\partial x} + \frac{\partial (h\, v)}{\partial y} \right].$$

- **update_velocities:**
  - Uses local η-values to compute velocity changes, e.g.

$$u_{i,j}^{n+1} = \left(1 - \gamma \Delta t \right) u_{i,j}^{n} - \frac{g\, \Delta t}{\Delta x} \left(\eta_{i,j} - \eta_{i-1,j} \right) .$$

Gathering and Output

- **MPI_Gatherv:** Rank 0 collects all subdomains' final η-, u-, and v-fields into global arrays.
- **Displacements and sizes:**
  - We compute each rank's portion size and displacement in the global buffer (using arrays like `recv_size_eta` and `displacements_eta`).



Conclusion for MPI Parallelization

- Create a 2D Cartesian topology to split the domain.
- Each process stores only a local subdomain plus ghost cells.
- Non-blocking communications (`MPI_Isend`, `MPI_Irecv`) exchange the necessary boundary information.
- Local updates of η, u, and v proceed in parallel.
- Rank 0 gathers and outputs the final results.
- This method *minimizes communication* by ensuring only boundary data is exchanged among neighbors.

OMP Parallelization

Serial code to OMP Parallelization code

- We used `#pragma omp parallel for` for our main loops in these function:
  - `interp_bathy()`
  - `update_eta()`
  - `update velocities ()`
  - `boundary conditions ()`(partially)
- As far as we tested, (considering only CPU code), using collapse was slowing down our result so we took the decision to not use them.

GPU Parallelization

GPU Parallelization

- **Data mapping:**
  - Fields like `all_data->eta`, `u`, `v` are transferred from CPU to GPU using `#pragma omp target teams distribute parallel for collapse(2) map(...)` in routines such as `update_eta()` and `update_velocities()`.
  - Example in `update_eta()`:

```
#pragma omp target teams distribute parallel for \
collapse(2)
map(tofrom: eta_gpu[0:nx*ny]) map(to: h_interp_gpu[], u_gpu[], v_gpu[])
```

- **Boundary conditions** and **source terms** also use `#pragma omp target` to update velocities or apply external forcing on the device.

GPU Parallelization

- We also define a custom mapper for our `data_t` struct to specify how arrays are moved to/from the GPU.
- During each timestep, certain arrays are copied *back* to the host for output:

```
#pragma omp target update from(*all_data->eta, *all_data->u, *all_data->v)
```

- This transfer ensures the most up-to-date GPU results are written to disk.

Nb of GPU	**Speedup**
1	1.33

Table 2: Speedup against serial code.

Tracing

Scaling Parameters

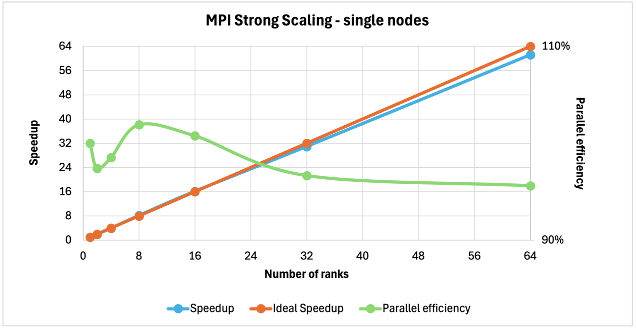
Following figures were computed using specific parameters to optimize resource utilization:

For the **strong scaling** `dx = 2, dy = 2, dt = 0.05 and max_t = 600.`

For the **weak scaling**: `dx = 5, dy = 5, dt = 0.02 and max_t = 600.` We halved until we reached: `dx = 0.3125, dy = 0.3125, dt = 0.02 and max_t = 600.`

MPI Scaling

Strong Scaling



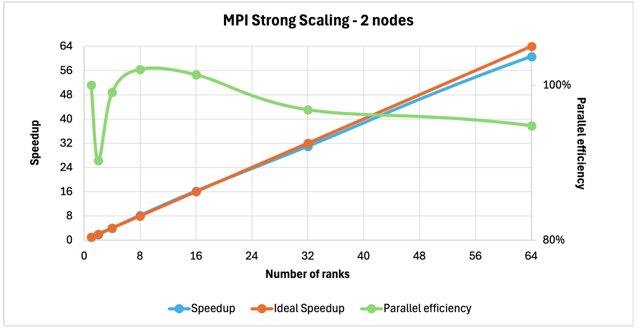
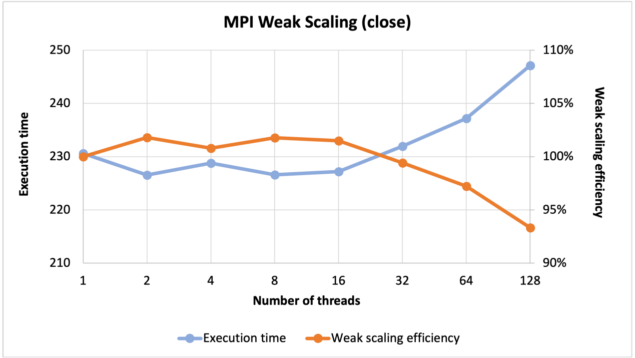


Figure 4: MPI strong scaling 1 and 2 nodes.

MPI Scaling (ii)

Weak Scaling



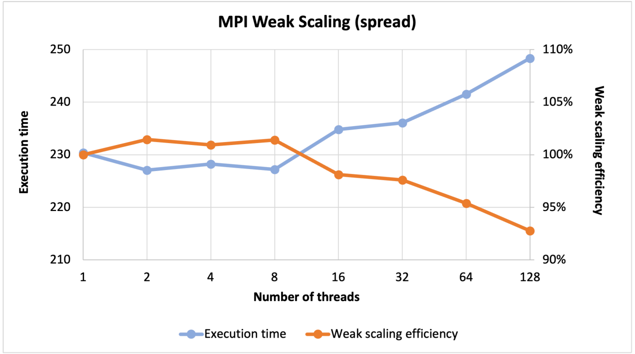
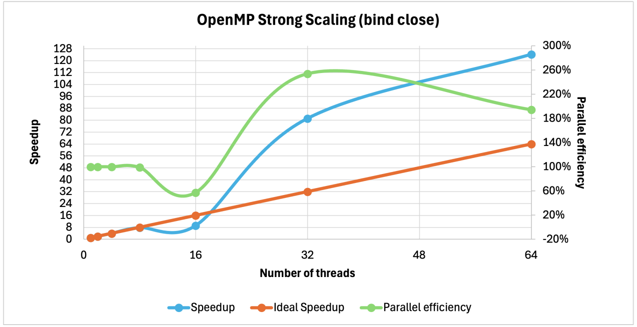


Figure 5: MPI weak scaling close and spread.

OpenMP Scaling

Strong Scaling



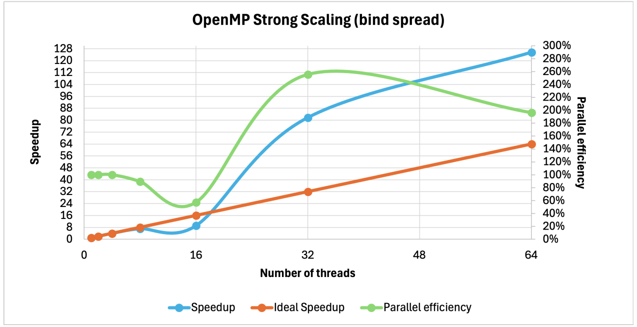
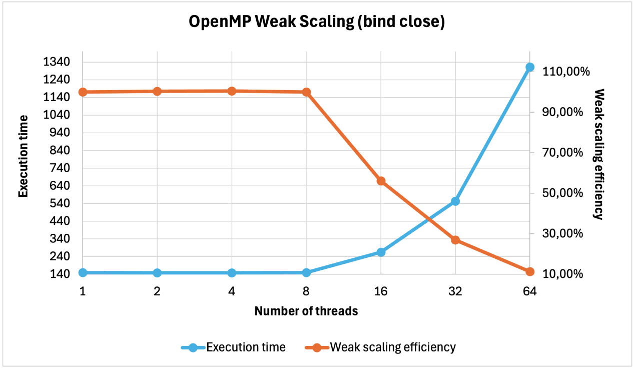


Figure 6: OpenMP strong scaling close and spread.

OpenMP Scaling (ii)

Weak Scaling



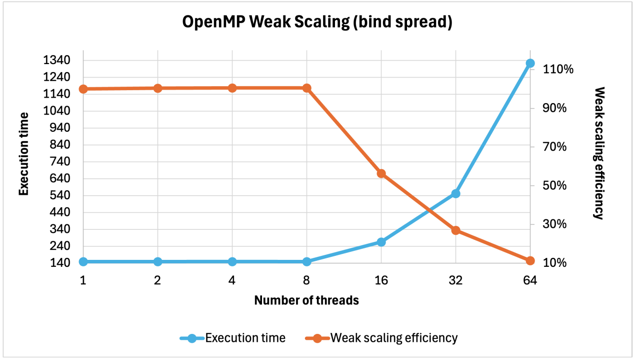
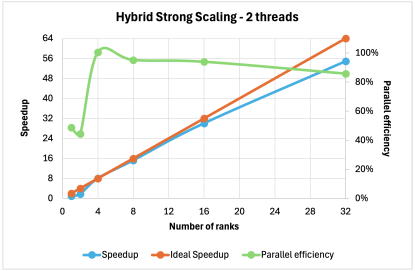
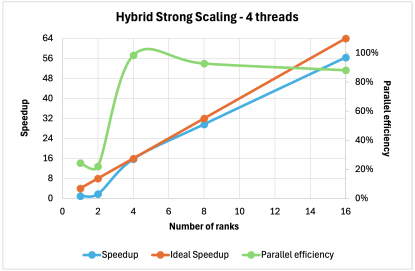


Figure 7: OpenMP weak scaling close and spread.

Hybrid Scaling

Strong Scaling





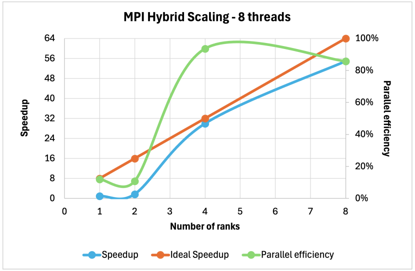
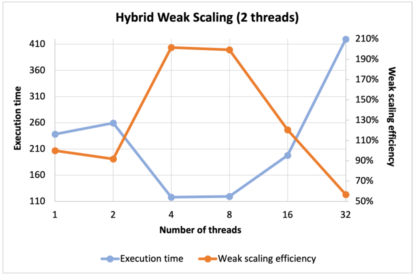
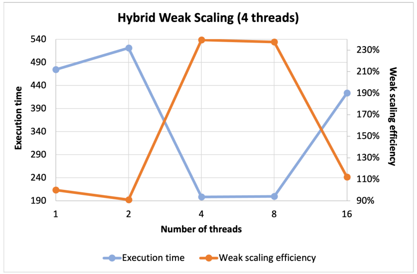


Figure 8: Hybrid strong scaling.

Hybrid Scaling (ii)

Weak Scaling





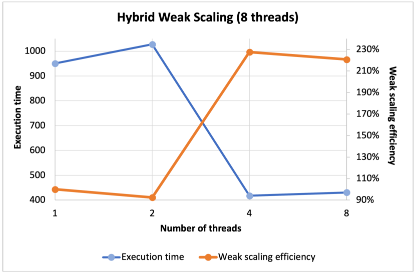
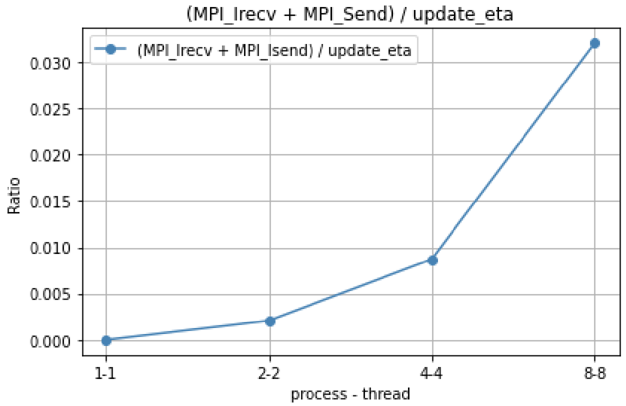


Figure 9: Hybrid weak scaling.

Hybrid Tracing

Eta analysis



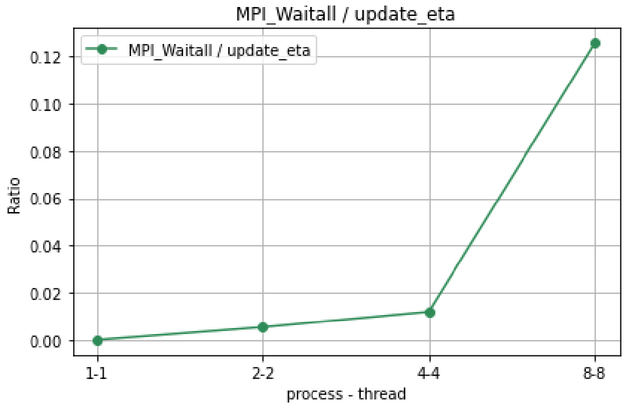
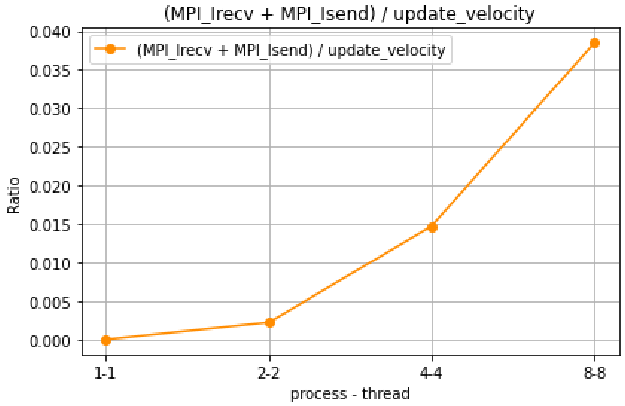


Figure 10: Eta tracing on hybrid code.

Hybrid Tracing (ii)

Velocities analysis



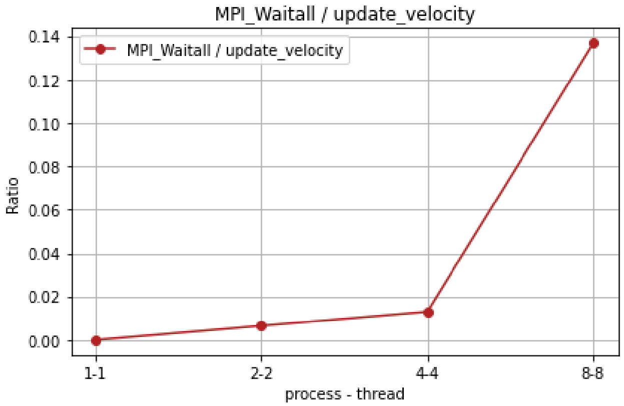
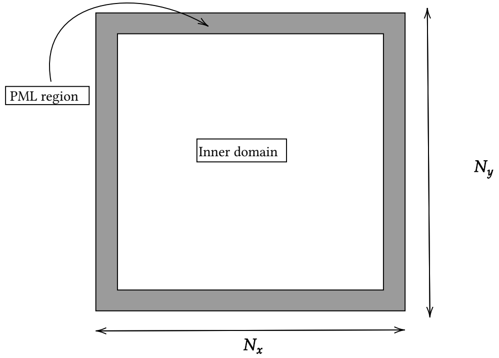


Figure 11: Velocity tracing on hybrid code.

Transparent boundaries

Principle

To simulate the "outer domain", the transparent boundaries may be implemented using the **Perfectly Matched Layer** (PML) method. Basically, we add a damping term in boundary regions:



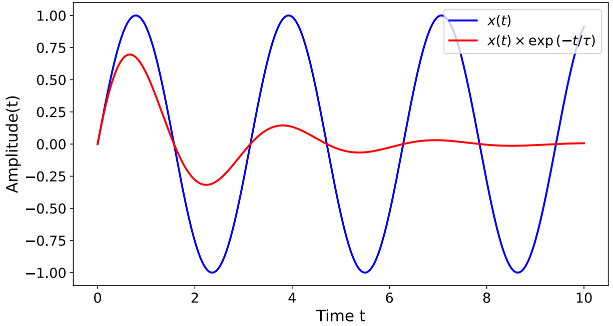


Figure 12: PML layer presentation and example on dummy function.

Calculation

The PML is applied to a distance relative to the boundaries:

$$\mathrm{PML}_{\mathrm{width}} \equiv d_{\mathrm{PML}} = \min\left(0.08N_x, 0.08N_y\right)$$

The damping amplitude coefficient is assumed dependent of the grid resolution:

$$\sigma_{\max} = 20.0\left(\frac{25.0}{\Delta \boldsymbol{x}}\right)$$

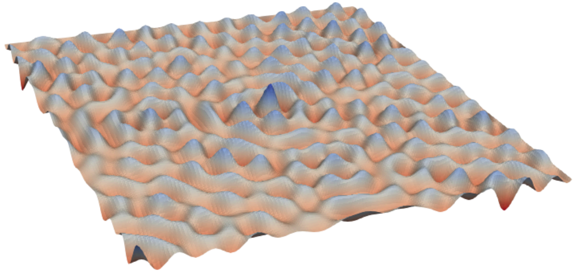
Cubic damping profile has been used:

$$\sigma = \max\left(\sigma_x, \sigma_y\right) \longrightarrow \sigma_{x,y} = \begin{cases} \sigma_{\max}\left(\frac{d_{\mathrm{PML}} - d_{x,y}}{d_{\mathrm{PML}}}\right)^3 \text{ if } d_{x,y} < d_{\mathrm{PML}} \\ 0 \text{ otherwise} \end{cases}$$

Final PML application: $\boldsymbol{u} \to \boldsymbol{u} \times \exp(-\sigma \Delta t)$

Results

Here is depicted the final timestep for the OMP/MPI code with and without the PML applied:



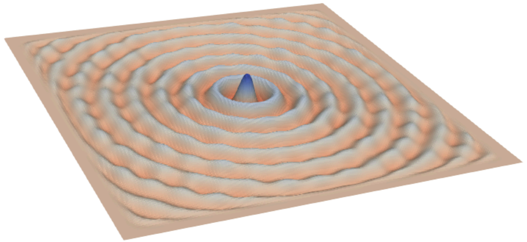


Figure 13: Boundary types comparison with $(\Delta x, \Delta y) = (25m, 25m)$; $\Delta t = 0.05s$ and $t_{\max} = 600s$. Reflective boundaries on the left and transparent boundaries on the right.

Coriolis Forces

Principle

The previously neglected Coriolis forces are now taken into account:

$$\frac{\partial \boldsymbol{u}}{\partial t} = -g\nabla\eta - \gamma\boldsymbol{u} \pm \text{f u}$$

Where:

- $f = 2\Omega\sin(\varphi) \rightarrow f_{\text{max}} = 2 \times 7.2931 \times 10^{-5}$ (Coriolis coefficient)
- $\Omega = \pi/12$ radians/hour (angular rotation rate of the Earth)

When discretized (example on u):

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} = -g\frac{\eta_{i,j}^{n+1} - \eta_{i-1,j}^{n+1}}{\Delta x} - \gamma u_{i,j}^{n} + f v_{i,j}^{n}$$

To account for this additional contribution and improve the model, we average actual coupled velocities as:

$$v_{i,j}^{n} \approx v_{\text{averaged}} = \frac{v_{i-1,j} + v_{i-1,j+1} + v_{i,j} + v_{i,j+1}}{4}$$

Assumptions

Physical Coriolis effects

Realistic Coriolis force ($f = 2 \times 7.2921 \times 10^{-5}$) becomes significant at:

- Spatial scale: L > 100 km
- Time scale: T > 17 h (inertial period at mid-latitudes)

Our simulation parameters Current numerical setup:

- Domain size: 160 × 160 points, dx = dy = 25m $\rightarrow$ 4km × 4km domain
- Time span: t_max = 600s (10 min), dt = 0.05s

Therefore, with scales 25 times too small spatially and 100 times too short temporally, three options exist:

1. Ignore Coriolis (f = 0)
2. Use realistic f (effect negligible)
3. Artificially increase f for testing $\rightarrow$ chosen

This amplification compensates for our reduced scales to observe and validate the implementation:

$$f_{\text{modified}} = 2 \times 7.2921 \times 10^{-2}$$

Results

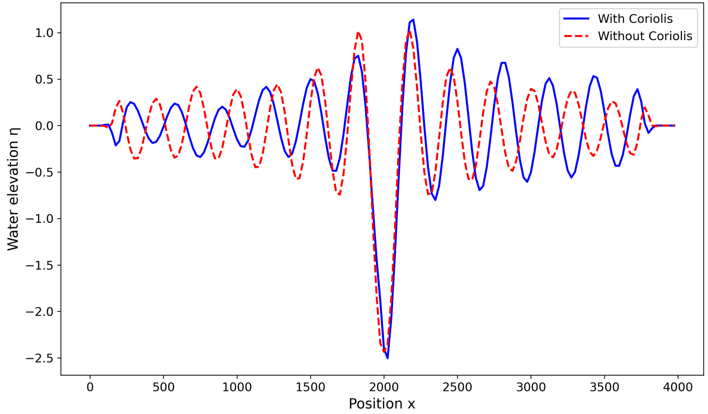


Figure 14: Water elevation profile at mid-domain.

After 10 minutes of simulation, we can observe modifications in the water elevation pattern: a decrease in water levels on the left side and an increase on the right side. This asymmetric distribution indicates the presence of a rotational motion induced by the Coriolis force.

Erratum

During code review, a critical issue was discovered:

- F_CORIOLIS was not properly defined in the code
- As a result, the Coriolis contribution was exactly zero