Parallel & Distributed Computing Project: Resolution of Dam Break 1D Problem using Saint-Venant Equations (OpenMP MPI)

Abdelmalek Bouaziz[1], Mohammed Machrouh[1] and Oussama Ziadi[1]
[1]College Of Computing, UM6P, Benguerir, Morocco

July 11th 2025

Abstract

In this project, parallel and distributed computing techniques were applied to a simple 1D dam-break solver written in C. The dam-break case is a classical benchmark problem commonly used to validate numerical methods and CFD solvers. The Rusanov scheme was employed to solve the Saint-Venant equations governing shallow water flow. Two parallelization approaches were implemented: one using OpenMP to parallelize loops within the solver, and another using MPI to decompose the domain into 1D chunks distributed across multiple processes. Benchmarking was conducted on an Intel Core `i9-14900HX` CPU with `16 GiB of DDR5` RAM operating at `5600MHz`.

1 Sequential Code Analysis

1.1 Code Structure

The sequential implementation of the `dam_break_rusanov.c` program simulates a one-dimensional shallow water dam-break problem using the Rusanov (local Lax-Friedrichs) scheme. The simulation computes fluid dynamics across discrete spatial cells for a series of time steps. The structure of the code is organized into the following logical blocks:

- **Initialization:** Parameters such as the number of spatial cells (`NX`), time step (`dt`), total simulation time (`TEND`), and gravitational constant (`G`) are defined. Arrays for water height (`h`), momentum (`hu`), velocity (`u`), and fluxes are allocated and initialized.
- **Initial Conditions:** The dam-break setup is configured with higher water height (5.0) on the left side of the domain (`X[i] < XM`) and lower (2.0) on the other, creating a discontinuity, as seen in Fig. 1.

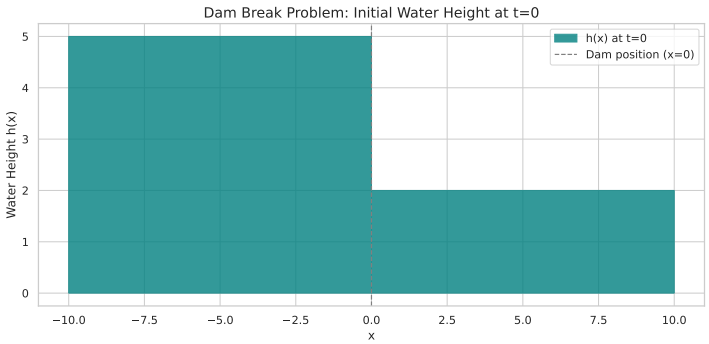

Figure 1: The Dam's Initial Condition

- **Time Stepping Loop:** The main simulation loop runs for a number of time steps determined by the total simulation time and time step size. Within each step:

- Fluxes are computed using the Rusanov scheme.
- The state variables (`h`, `hu`) are updated using finite-volume discretization.
- Boundary conditions are applied to maintain physical constraints.

- **Output:** Finally, final values of cell positions, height and speed are written to a file `output.dat`, which can be used for visualization or verification.

2 Version 1: Loop parallelization using OpenMP

The parallelization of the dam break simulation using the Rusanov scheme was done through the application of OpenMP directives to exploit shared-memory parallelism. The primary parallelization targets were the **max speed calculation**, **flux calculation**, **conservative variable update** and **Final values update** loops. Fig. 3 shows in detail the whole program workflow, along with parallelized loops. For the ones in red, we chose to not parallelize as it would cause communication overhead.

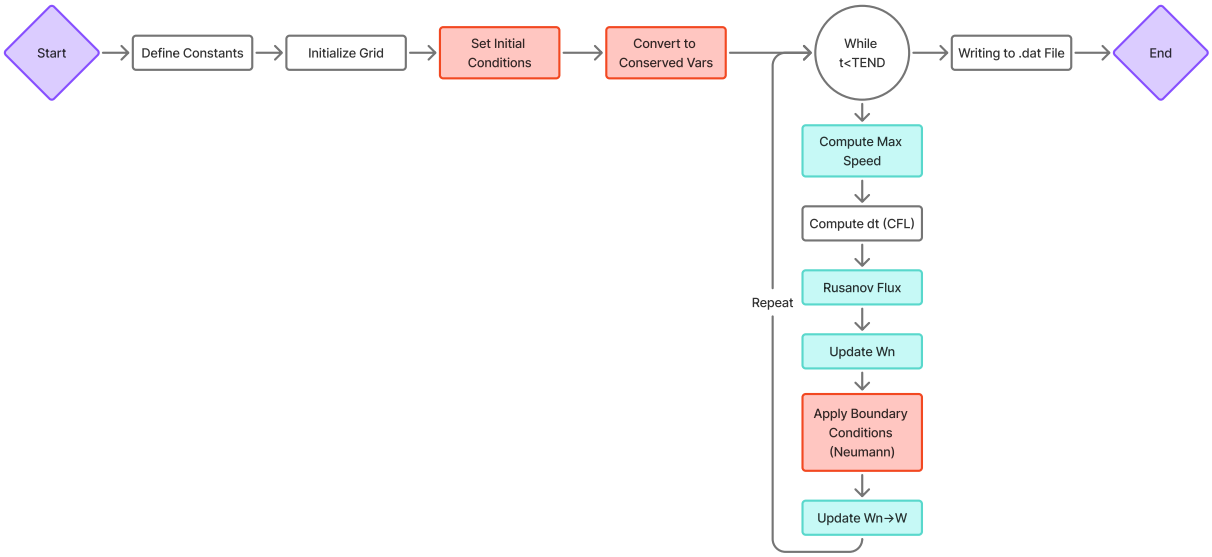

Figure 2: Code Workflow; Blue Loops are parallelizable, Red are not (No need to parallelize)

2.1 Parallelization Strategy

Each of the parallelized loops operates on independent elements of arrays, making them suitable for data-parallel execution with OpenMP. The parallelization was implemented by inserting `#pragma omp parallel for` directives before these loops, thereby distributing loop iterations across multiple threads. The key considerations were ensuring thread safety, maintaining correct memory access patterns, and minimizing synchronization overhead.

2.2 Details of Parallelized Loops

- **Max Speed Calculation:**

```
#pragma omp parallel for reduction(max:max_speed)
for (int i = 0; i < nc; ++i) { ... }
```

  This loop calculates the maximum wave speed across the domain for determining a stable time step. A reduction clause is used to correctly compute the maximum value in parallel.

- **Flux Calculation:**

```
#pragma omp parallel for
for (int i = 0; i < nc - 1; ++i) { ... }
```

Fluxes at the cell interfaces are computed independently, based on neighboring values. Since each iteration only reads from shared arrays and writes to its own index, no race conditions occur.

- **Conservative Variable Update:**

```
#pragma omp parallel for
for (int i = 1; i < nc - 1; ++i) { ... }
```

The update of water height and momentum is also parallelized, leveraging the locality of finite-volume operations. Each thread safely updates separate elements of the state arrays.

- **Updating the main Variable:**

```
#pragma omp parallel for
for (int i = 0; i < nc; ++i) {
        W[0][i] = Wn[0][i];
        W[1][i] = Wn[1][i];
    }
```

3 Version 2: Domain Decomposition using MPI

The second parallelization strategy employed domain decomposition using the Message Passing Interface (MPI). This approach partitions the global spatial domain, represented by `nc_total` computational cells, into smaller subdomains, each managed by a dedicated MPI process.

3.1 MPI Initialization and Domain Distribution

At the beginning of the program, the MPI environment is initialized using `MPI_Init`. Each executing instance becomes an MPI process, obtaining a unique identifier (`rank`) and the total number of processes (`num_procs`) via `MPI_Comm_rank` and `MPI_Comm_size`, respectively.

The global domain of `nc_total` cells is then divided among these processes. The number of real cells assigned to each process, `nc_local`, is calculated to ensure an almost even distribution, with the first `remainder` processes receiving one additional cell.

```
int nc_base = nc_total / num_procs;
int remainder = nc_total % num_procs;
int nc_local = nc_base + (rank < remainder ? 1 : 0);
```

Crucially, each process also determines its `global_start_idx`, representing the global index of its first local cell. This allows each process to correctly switch from and back to the global domain.

```
int global_start_idx = rank * nc_base + fmin(rank, remainder);
```

Unlike the sequential version which uses a single global array for the entire domain, the MPI implementation distributes the computational load. Each process then allocates its local arrays (e.g., `h_local`, `u_local`, `W_local`) with `nc_local + 2` elements. These additional two elements serve as **halo cells** (or **ghost cells**), one on each side, which are essential for inter-process communication.

3.2 Local Initialisation of Conditions

The `init_conditions_local` function is responsible for setting the initial state of the fluid for each process's local subdomain. It iterates over the `nc_local` real cells assigned to the process. For each cell, it computes its global left interface coordinate:

```
double current_x_left_interface_global = XLEFT_GLOBAL + (global_start_idx + i) * dx;
```

This coordinate is then used to apply the initial dam break conditions (height `HLEFT` or `HRIGHT`, and zero velocity). To ensure exact reproducibility of the sequential version's initial conditions, the discontinuity check is based on this *left interface* of each cell, mimicking the original implementation's use of `x[i]`. The `xc_local` array, used for output, still stores the cell centers.

It is important that the initial values are placed correctly within the local arrays: `h_local[i + 1]` and `u_local[i + 1]` are used because index 0 is reserved for the left halo cell. After initialization, these values are copied into the main conservative variable array `W_local` for the real cells.

3.3 Communication between Halos (Ghost Cells)

Numerical schemes, such as Rusanov, are inherently local, requiring data from adjacent cells to compute fluxes and update cell values. In a distributed memory environment, this means that cells located at the boundaries of a process's local subdomain need information from cells belonging to neighboring processes. This information is exchanged through **halo cells**.

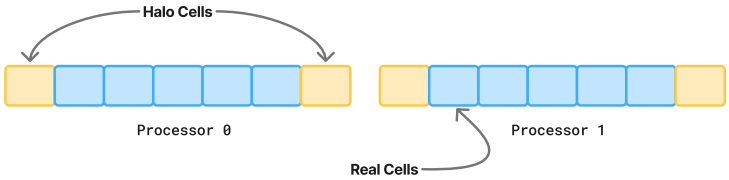

Figure 3: Halos cells vs Real cells

At the beginning of each time step, after the `W_local` array has been updated from the previous iteration, an exchange of halo data is performed using `MPI_Sendrecv`. This collective operation is crucial as it combines a send and a receive operation into a single atomic call, preventing potential deadlocks that could arise from separate send/receive calls.

For each pair of communicating processes (e.g., `rank` and `rank+1`):

- A process sends the values of its outermost real cells to its neighbor.
- Simultaneously, it receives the corresponding values from its neighbor, storing them in its designated halo cells.

A key challenge encountered during development was the handling of the `W_local[2][...]` static 2D array. Due to its non-contiguous memory layout (where `W_local[0]` and `W_local[1]` are separate blocks of memory), attempting to send both conservative variables (`h` and `hu`) in a single `MPI_Sendrecv` call with a count of 2 resulted in corrupted data. The solution involved performing **separate `MPI_Sendrecv` calls for each variable** (height `W_local[0]` and momentum `W_local[1]`) for each direction (left/right). This ensures that only contiguous blocks of memory are sent, correctly filling the halo cells. Unique communication tags (0, 1, 2, 3) are employed to differentiate these four distinct communication patterns (send/receive for H/HU in left/right directions).

After the halo exchange, the `h_local` and `u_local` arrays are fully updated, including their halo cells, from the (now current) `W_local` values. This prepares them for the Rusanov flux calculation in the subsequent step.

3.4 Global Time Step Synchronization

For the simulation to remain stable and accurate, the Courant-Friedrichs-Lewy (CFL) condition must be satisfied across the entire domain. This condition dictates that the time step (`dt`) must be small enough to ensure that information does not propagate more than one cell per time step. This requires the use of the global maximum characteristic speed.

Each process first calculates its `max_speed_local` based on the fluid properties within its subdomain. Then, the `MPI_Allreduce` collective operation is used to determine the `max_speed_global` from all local maximums. This operation ensures that every process receives the overall maximum speed, and consequently, all processes compute and use the exact same time step `dt` for the current iteration. This synchronization is vital for the global stability of the explicit time-marching scheme.

3.5 Local Computations and Boundary Conditions

Once the time step `dt` is determined and halos are updated, each process proceeds with its local computations:

- **Rusanov Flux Calculation:** The `rusanov_flux` function is called locally. It computes the numerical fluxes across all interfaces within the process's extended subdomain, including those involving halo cells.
- **Update of Conservative Variables:** The updated conservative variables (`Wn_local`) are calculated for the real cells based on the fluxes and the previous time step's values.

Finally, the Neumann (zero-gradient) boundary conditions are applied. These global boundary conditions are only handled by the processes responsible for the physical edges of the entire domain:

- `rank 0` applies the condition at the global left boundary, setting the first real cell's value to that of its neighbor (`Wn_local[j][1] = Wn_local[j][2]`).
- The last process (`rank == num_procs - 1`) applies the condition at the global right boundary, setting its last real cell's value to that of its inner neighbor (`Wn_local[j][nc_local] = Wn_local[j][nc_local - 1]`).

After all calculations for the current time step are complete, the updated values in `Wn_local` are copied back into `W_local`, preparing the system for the next time iteration.

3.6 Gathering of the Results

Upon completion of the full simulation up to `TEND`, the final height (`h`) and velocity (`u`) values from each process's local subdomain must be collected onto a single process (the root process, `rank 0`) to generate the complete solution file.

Due to the possibility of `nc_local` varying slightly between processes (if `nc_total` is not perfectly divisible by `num_procs`), the `MPI_Gatherv` (Gather Variable) collective operation is employed. This operation allows the root process to receive data segments of varying sizes from each contributing process. Before the gathering, each process prepares a `h_send_buffer` and `u_send_buffer` containing only its real cell data (excluding halos).

The root process first collects the `nc_local` value from all processes using `MPI_Gather`. It then uses this information to calculate the `recvcounts` (how many elements to expect from each process) and `displs` (the starting offset for each process's data in the global array). Finally, `MPI_Gatherv` is invoked to assemble the complete `h_global`, `u_global`, and `xc_global` arrays on the root process.

The root process (`rank 0`) then measures the total execution time using `MPI_Wtime()` and writes the collected global solution to the `output_mpi.dat` file, containing `x`, `h(x)`, and `u(x)` values. All dynamically allocated memory is then freed by the respective processes before `MPI_Finalize()` is called to clean up the MPI environment.

4 Results

4.1 OpenMP parallel version vs Sequential

4.1.1 Solution Verification

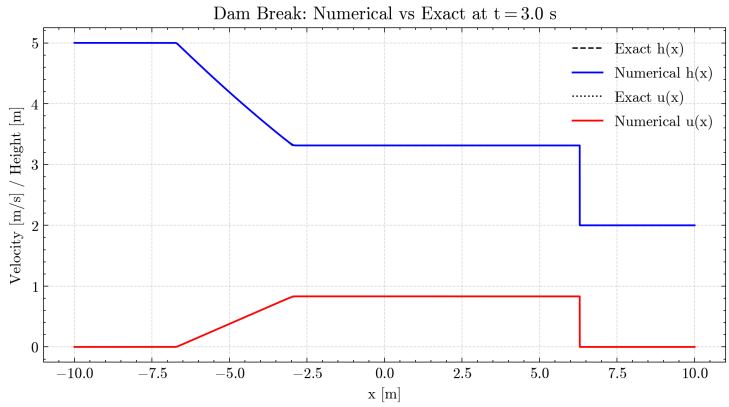

Figure 4: Exact Solution (dashed) vs Numerical Solution (solid)

The solution plot (Fig. 4) shows a classic dam-break profile with a sharp discontinuity and shock propagation. The OpenMP parallelization preserves the numerical accuracy of the Rusanov scheme. There is no visible degradation in solution quality, confirming that parallel execution does not introduce artifacts or instability.

4.1.2 Execution Time (Strong Scaling)

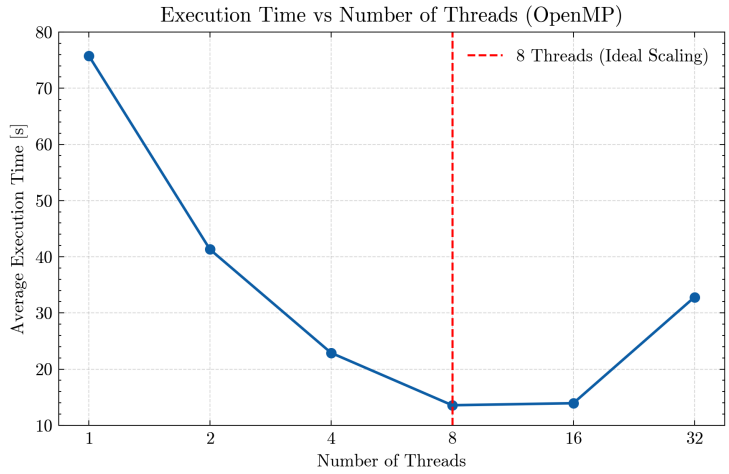

Figure 5: Execution Time vs Number of Threads

The execution time decreases significantly as the number of threads increases from 1 to 4 (Fig. 6), and continues to improve up to around 8 threads. Beyond this, the gains of parallelization diminish, which indicates either memory bandwidth limits or parallel overhead. As we know, this trend is typical of shared-memory systems where at one point, more threads limit scalability.

4.1.3 Parallel Efficiency

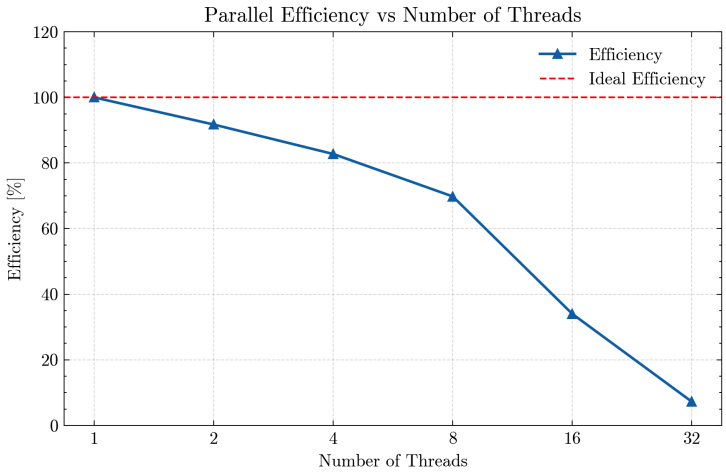

Figure 6: Parallel Efficiency vs Number of Threads

Efficiency remains above 80% for up to 4 threads, gradually decreasing with more threads. At 16 threads, efficiency drops to around 40–50%, showing that the fixed problem size leads to underutilization of resources at higher core counts. This suggests the code is well-suited for moderate core parallelism but not ideal for many-core systems without further optimization (e.g., domain decomposition or dynamic scheduling).

4.1.4 Speedup

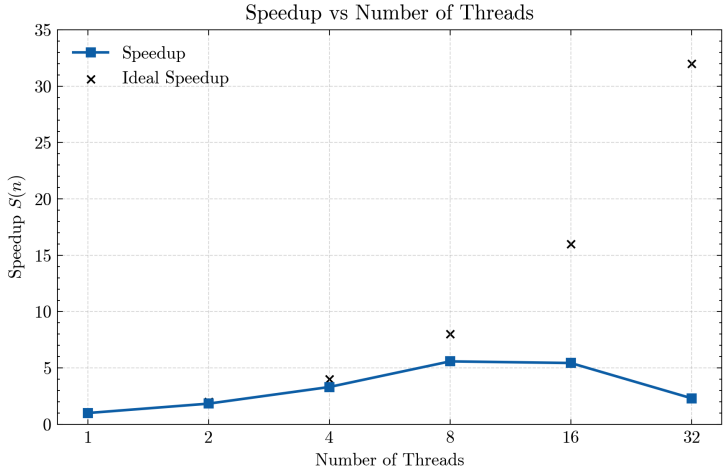

Figure 7: Speedup vs Number of Threads

The speedup curve initially grows nearly linearly with the number of threads (up to 4–8), which confirms effective loop-level parallelization. However, speedup plateaus beyond 8 threads, indicating diminishing returns due to Amdahl's Law and synchronization overheads.

4.2 MPI version vs Sequential

4.2.1 Execution Time (Strong Scaling)

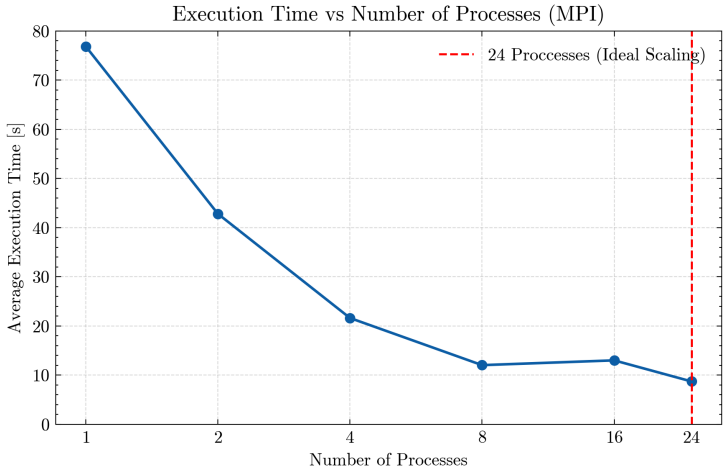

Figure 8: Execution Time vs Number of Threads

The execution time decreases significantly as the number of threads increases from 1 to 4 (Fig. 9), and continues to improve up to around 24 threads. Much more performant than OpenMP version.

4.2.2 Parallel Efficiency

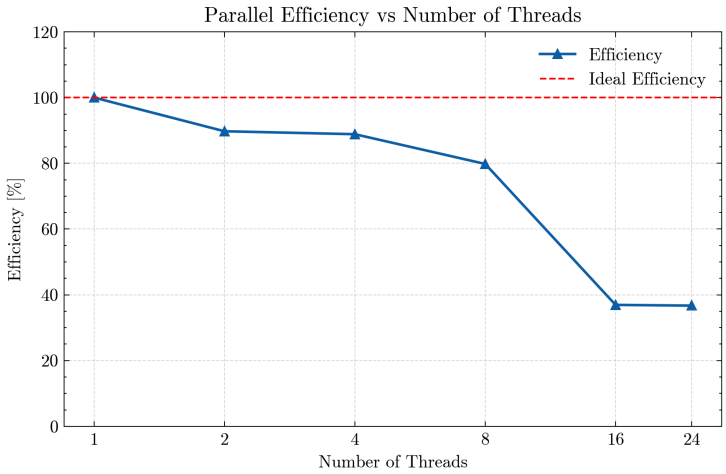

Figure 9: Parallel Efficiency vs Number of Threads

Efficiency remains above 80% for up to 8 threads (way better than OpenMP), gradually decreasing with more threads. At 16, 24 threads, efficiency drops to around 40%, and stays static as the number of threads increase.

4.2.3 Speedup

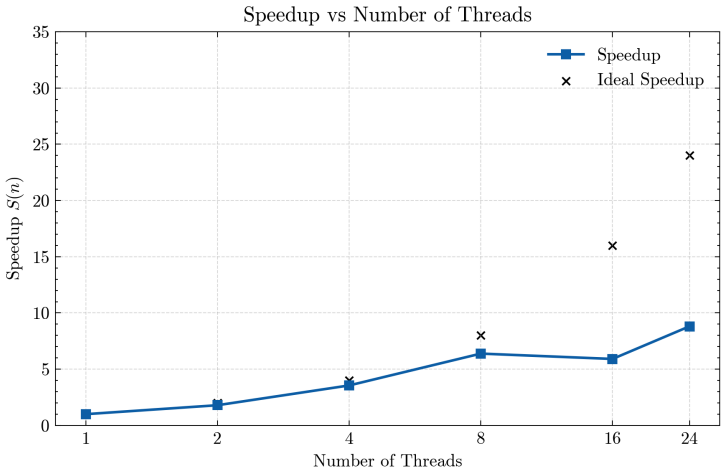

Figure 10: Speedup vs Number of Threads

The speedup curve initially grows almost linearly with the number of threads (up to 34), which confirms effective loop-level parallelization. Unlike the first version which stagnated after 8 threads.