INFO 0939 Project 2024

Last updated: 24/09/2024

Goals of the Project

- Solve a physical problem modelled by partial differential equations using a finite difference scheme coded in C.
- Parallelize the code on distributed memory systems using MPI, and on shared memory systems using OpenMP.
- Combine MPI and OpenMP to make the most of a supercomputing cluster.
- Test the acceleration potential of modern GPUs using OpenMP.
- Learn to profile the code and run weak and strong scalability analyses.
- Explore one or more advanced topic(s) based on your own interests.

Project statement

Wave propagation at the surface of the ocean can be modelled using the so-called "shallow water" equations, which are derived by depth-integrating the Navier-Stokes equations. This assumes that the horizontal velocity field is constant throughout the depth of the ocean, allowing to elimitate the vertical velocity from the equations. The linear shallow water model for which we neglect the Coriolis force writes:

$$\frac{\partial \eta}{\partial t} = -\nabla \cdot (h\mathbf{u}) \tag{1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -g\nabla\eta - \gamma\mathbf{u} \tag{2}$$

where the unknown fields $\eta(t,x,y)$ and $\mathbf{u}(t,x,y) = (u(t,x,y), v(t,x,y), 0)$ are respectively the free-surface elevation and the depth-averaged velocity, and where

- $h(x,y)$ is the depth at rest;
- g is the gravitational acceleration;
- γ is the dissipation coefficient.

The first equation is the continuity equation and the second is derived from the momentum equation.

The domain of study is a rectangle whose dimensions are inferred from a bathymetric map (map of $h(x,y)$), provided in a binary file (see appendix).

Discretization in space is performed as depicted in Figure 1: equation (1) is discretized at the center of the cells while equation (2) is discretized at their boundaries, both using finite differences (see the appendix). Note that the size of these cells usually does not match (is typically much smaller) than

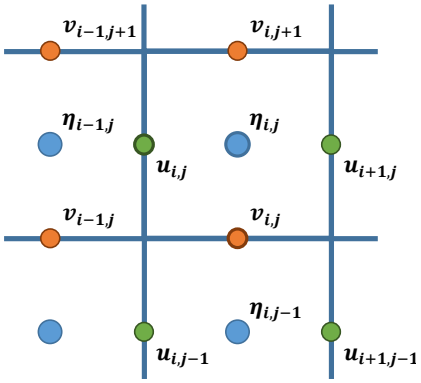

Figure 1: Diamond scheme for spatial discretization. The elevation η is computed in the center of the cells while the components of the depth-averaged velocity u and v are computed on the boundaries of the cells.

the resolution of the bathymetric map, and interpolation of the bathymetric data is necessary on the finite difference grid points.

Discretization in time is decoupled: η is computed at time $n\Delta t$ while $\mathbf{u}$ is computed at time $(n+\frac{1}{2})\Delta t$ with $n = 1, 2, \ldots$.

The initial condition is a zero free-surface elevation and zero velocity, *i.e.* $\eta^0_{i,j} = u^0_{i,j} = v^0_{i,j} = 0$, $\forall (i,j)$. Note that i and j do not have exactly the same range for the three unknown fields.

Several boundary conditions can be considered. By default, the normal velocity is fixed to zero (impermeable boundaries) on the left, bottom and right sides of the domain,

$$\begin{cases} u^n_{0,j} = 0 \\ u^n_{i_{\max},j} = 0 \\ v^n_{i,0} = 0 \end{cases} \qquad \forall i, j;$$

while on the top of the domain the v component of the velocity is imposed to

$$v^n_{i,j_{\max}} = A \sin(2\pi f t), \quad \forall i, \tag{3}$$

where $t = (n+\frac{1}{2})\Delta t$. Condition (3) models a simple, steady-state wave source of fixed amplitude, and can be used e.g. with the bathymetry file `h_simple.dat`.

Instructions

A serial C code that solves the discretized equations using an explicit time integration scheme is available on the NIC5 cluster: see `https://people.montefiore.uliege.be/geuzaine/INFO0939/project/`. The program takes as a single argument a parameter file, whose format is described in the appendix, and creates output files that can be read with Paraview (`https://paraview.org`).

In a first project phase (deadline: 26/11/2024) you are asked to:

1. Improve the evaluation of the bathymetry using bilinear interpolation (in the serial code the evaluation is done using a nearest-neighbor search).

2. Experimentally study the stability of the explicit time integration scheme, for the constant bathymetry provided in the `h_simple.dat` file.

3. Perform an analysis of the arithmetic intensity of the method and determine the main limiting factor for the speed of your algorithm: is the code memory bound or CPU bound? What do you expect if you run the code on a machine capable of 200 GB/s of memory bandwidth and 2.8 TFLOPS/s of processing power?

4. Evaluate potential bottlenecks in the serial code, based on the lecture on CPU cache hierarchies.

5. Parallelize the serial code using MPI, by subdividing the domain into $P_x \times P_y$ partitions, where the positive integers P_x and P_y designate the number of partitions along x and y, respectively, and are chosen to minimize the amount of MPI communications.

6. Parallelize the serial code using OpenMP. Could you advantageously make use of the `collapse` clause?

7. Integrate the OpenMP parallelization strategy into the MPI code to produce a hybrid MPI+OpenMP code that can efficiently target the cluster architecture of the NIC5 supercomputer.

8. Perform a scalability analysis of the MPI, OpenMP and MPI+OpenMP codes, by evaluating both the strong and the weak scaling. Choose appropriate parameter values for these scaling runs (the provided `param_simple.txt` is not appropriate!), deactivate outputs to disk (by setting the sampling rate to 0), make sure to reserve appropriate resources on the clusters (e.g. full nodes), and perform measurements multiple times to reduce uncertainty.

9. Using the tools presented during the lectures (Score-P, Scalasca, Cube), explain the results of the scalability study. (Never run the scalability studies themselves within the profiling tools, as the profilers add significant overhead!)

10. Accelerate the serial code using OpenMP on a single GPU of the Lucia supercomputer. Compare and analyze the performance of the OpenMP CPU and GPU codes.

In second second phase (deadline: 20/12/2024), you are asked to explore **at least one** of the following topics:

1. Run and analyze large scale simulations of physical relevance, by creating one or more realistic bathymetry files and specifying relevant sources.

2. Combine MPI and OpenMP for multi-GPU parallelization and perform a scalability analysis on multiple GPUs.

3. Extend the physical model by adding Coriolis forces and more sophisticated boundary conditions to simulate a transparent boundary.

4. Implement an implicit variant of the time integration scheme, using a linear solver of your choosing, and compare the performance of the explicit and implicit approaches on relevant benchmarks.

Appendix

Discretized equations

Using an explicit Euler scheme, the discretized equations read as follows:

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

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

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

where $\mathbf{x}^{u_{i,j}} = (x^{u_{i,j}}, y^{u_{i,j}})$ is the location of the u component of the velocity (resp. for the v component):

$$\mathbf{x}^{u_{i,j}} := (x_a + i\Delta x,\ y_a + (j+\frac{1}{2})\Delta y)$$

$$\mathbf{x}^{v_{i,j}} := (x_a + (i+\frac{1}{2})\Delta x,\ y_a + j\Delta y)$$

Using implicit Euler instead would lead to evaluating all the velocities (u and v) at step $n+1$ instead of n in the right-hand-sides of the first three equations.

Interpolation of the depth

Let (x_i, y_j) denote the location of the bathymetric data and δx and δy the spatial steps between two data points. To determine the depth at a point (x,y), you should find the couple (k,l) such that $x_k \leq x < x_{k+1}$ and $y_l \leq y < y_{l+1}$ and use the bilinear interpolation between the four surrounding data:

$$\begin{aligned} h(x,y) = \Big[& h_{k,l}\ (x_{k+1}-x)(y_{l+1}-y) \\ & + h_{k+1,l}\ (x-x_k)(y_{l+1}-y) \\ & + h_{k,l+1}\ (x_{k+1}-x)(y-y_l) \\ & + h_{k+1,l+1}\ (x-x_k)(y-y_l) \Big] / (\delta x\, \delta y). \end{aligned}$$

Parameter file format

The input parameter file is an ASCII text file that contains the simulation parameters in the following order:

```
dx
dy
dt
max_t
g
```

```
gamma
source_type
sampling_rate
input_h_filename
output_eta_filename
output_u_filename
output_v_filename
```

with

- `dx`, `dy` (double): spatial grid steps Δx and Δy;
- `dt` (double): time step Δ_t;
- `max_t` (double): maximum simulation time; the maximum number of time steps is $\lfloor$`max_t`$/\Delta_t \rfloor$;
- `g` (double): gravitational acceleration g;
- `gamma` (double): dissipation coefficient γ;
- `source_type` (integer): specification of the source. A value of 1 corresponds to the default boundary condition on the vertical velocity (3). Other values are free for you to define in your implementation.
- `sampling_rate` (int): sampling rate at which output files should be created (0: never save, 1: save all steps, 2: save 1 step out of 2, etc.);
- `input_h_filename` (string): name of the input bathymetric file containing the values of h on a grid (the file format is described in "Binary data file format" below);

Binary file format

Bathymetric data is provided in a binary file with the following format:

```
nx ny dx dy h...
```

with

- `nx` (int): number of nodes along x;
- `ny` (int): number of nodes along y;
- `dx` (double): spatial grid step along x;
- `dy` (double): spatial grid step along y;
- `h...` (array of `nx`$\times$`ny` doubles): the values are assumed to be given by row; see the `read_data()` and `write_data()` functions in the sequential code.

General remarks on C coding style

Your coding style will be evaluated. As such, some quality in the submitted code is expected, and you should strive to have a clean and neat coding style. In general, think about who will read your code and ask yourself if it is clear and understandable. Please beware of some common mistakes that will lead to penalties on your final grade:

- If you `malloc()` something, remember to `free()` it.
- Check return values for failure, especially from `malloc()`, `fopen()` or similar calls. Don't assume that those calls will always succeed.
- Don't use Variable Length Arrays, i.e.

  ```
  int function(int N) {
      int array[N];
  }
  ```

  The value of `N` must be known at compile time. If it is not the case, use `malloc()`.
- Don't abuse comments. If something is obvious, don't comment it. If you feel the need to comment some code, ask yourself if perhaps you can write the code in a clearer way instead of putting that comment.
- Try to use relevant and appropriate variable and function names. Be coherent in naming things, in order to make it easy to track down where data goes.
- Whenever possible avoid global variables: they are one of the factors that make it harder to make your code thread safe.
- Limit the length of your functions: if a function does not fit on your screen, perhaps you should split it in smaller pieces. Also, if you have a function that takes more than 10 parameters, perhaps you need to think if you really need all of them, or it might be time to create a `struct`?