Assignment 3

High Performance Parallel Computing 2022

Kimi Cardoso Kreilgaard & Linea Stausbøll Hedemark

March 2022

1 OpenACC parallelise the program

Since we weren't able to access ERDA's GPUs, we wrote "`module load nvhpc-nompi`" in the terminal, which to our understanding means that while we are still running on a CPU we can construct the architecture of the code into something that would work with a GPU. Since we are only working on the CPU, it will contain all the memory, where if we worked with a real GPU it would have its own seperate memory, meaning that we would have to be careful with the end results, making sure that before the CPU main thread stores any results it needs to be updated with the results obtained on GPU. We tried to be aware of this when developing the code, but if something works on the CPU environment we are in, we cannot be totally sure it would also work on a GPU environment. Our ability to test the solution is therefore limited. We were also not able to use the suggested profiler to indicate to us where to begin the optimization of the code performance. Instead we went with our instincts and parallelised were we thought it made sense. We focused on parallelising parts of the code that performs what is called "stencil operations", operations where each element of an output array depends only on a smaller region of the input array, since these generally perform better on GPUs. There are many functions within functions in this segment, so here we will go through step by step (chronologically according to how the code is executed when compiled not according to the order the code is written in text) how we inserted the pragmas to parallise the code.

- In the `simulate` the `water_world` class is initialized on the main thread, the CPU. Before we can delegate any work to the "pretend GPUs" we have to make the data available to them. We do this with the command `simulate` before any work is done.

- Within the `integrate` function, the two functions `exchange_horizontal_ghost_lines` and `exchange_vertical_ghost_lines` are used. They each consist of a loop which we can parallise on the "pretend GPUs" with the command: `#pragma acc parallel loop present(data)`. Here the `data` parameter refers to the input to the function which is either `w.e` or `w.v`, both attributes of the `water_world` class. The clause `present` tells the given "pretend GPU" that it already has the data available to it, and doesn't need to copy it from the CPU.

- Next here are two sections of nested loops in the `integrate` function which are parallised with the command: `#pragma acc parallel loop collapse(2) present(w.u,w.v,w.e)`. `collapse` rewrites the loop into one, and `present` again tells the "pretend GPU" that no data has to be copied to perform the operation.
- Now that all operations are performed the result (the updated `water_world`) is stored in the grid `water_history`, an array belonging to the CPU. This is done within the `simulate` function. We get the right result on our setup, by not changing anymore. But if we were to run it on actual GPU's the memory would not be shared and we would not be storing the updated attributes of `water_world.e`. This means we have to update the data on the CPU and we do this by inserting the command: `#pragma acc update self(water_world.e)`.

2 Strong and weak scaling

As we weren't able to run our program on actual GPUs, our elapsed time increased after we added pragmas. So making a plot of weak and strong scaling would not make sense based on the results of our program. Instead we've included an example of what could be expected, see fig. 1. Strong scaling is defined as how the solution time varies with the number of processors for a fixed total problem size. More specifically, the plot shows the speedup as a function of number of processers used. The problem is fixed by keeping the gridsize and number of iterations constants. There are different concepts in `open ACC` that can speed up the process: workers, vectors and gangs. How much work can be done by the worker is limited by his speed, and a single worker can only move so fast. Each worker has a vector, which can change the speed of the worker. A larger vector means that the operations can be performed more places at once, which would make the worker faster. Gangs are groups of workers, where each gang can share some ressources making them work together better, while different gangs can work independently of each other. When we want to perform strong scaling we thus have to be sure that we are only increasing one thing, which can be on the x-axis symbolising number of processors. Since we didn't get the actual GPU to work, this was hard to get any hands on understanding of. We were left with the hint that when using pragmas in strong scaling we should add `num_gangs(n)` and `vector_length(64)`, where n could be any number between 1 and 100. Our understanding is that what we would vary and measure the speedup against is the number of workers, while `num_gangs(n)` ensures that we always have the same number of groups and `vector_length(64)` or another number, ensures that a worker always has the same "equipment" to perform the job. The result for strong scaling would ideally be a linear relationship, but this is practically never possible since there is always a part of the code that is sequential and thus not able to be parallelised. Weak scaling describes how the solution time evolves depending on the number of processors for a fixed problem size per processor. For weak scaling it is thus important to increase the overall problem in our program, such that the problem size is fixed for all processors. In this case it would mean increasing the grid, so the grid portion each processor would operate on is constant. The speed up for weak scaling is defined as: $speedup(N) = t(1)/t(N) \cdot (work\_load(N)/work\_load(1))$. Where the last term scales the speed up according to the increased workload for a larger number of threads.

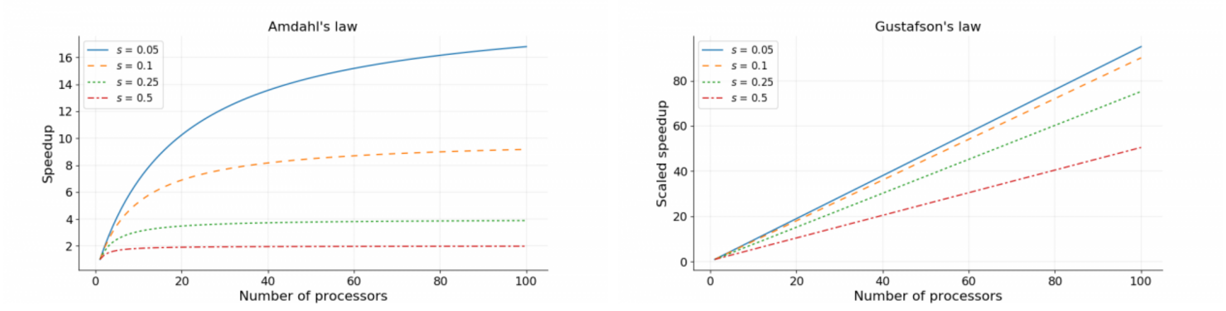

Figure 1: Example plots of strong and weak scaling, borrowed from https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/.

But again, as we weren't able to run on GPU and the performance didn't get better, our measurements of strong and weak scaling don't show the expected behaviour at all.

3 Physics

The theoretical prediction is $c = \sqrt{gH}$, where the elevation, H, is set to 1m in our program, so the phase speed is simply the squareroot of the gravitational acceleration, and we expect a phase speed at around $c \approx 3.13\frac{m}{s}$. To determine a numerical estimate we assume that all distance units are in meters (so the grid we are simulating in is 512×512 square meters) and all time units are in seconds. This means that with 1000 iterations and a time step of $dt = 0.05$ s, we are simulating a total of 50 seconds. To determine the phase speed we need to keep track of the distance travelled by the wave, this is easiest to identify by the movement of the maximum elevation. Since the geometry of this wave is symmetrical, we can reduce the problem to one dimension, by looking at a single slice of the elevations. We thus fix the y-coordinate of the grid to be 256, which gives us a slice in the middle. Furthermore since the problem is also symmetrical about the x-axis, we only look at the part of the wave that propagates to the right. This ensures that when we take the maximum value of the elevation, we don't flip back and forth between the left and right propagating waves between time steps, which might happen if one side is slightly bigger than other, due to numerical fluctuations. This process of selecting which data to analyse is visualized in Fig. 3. For each time step saved to the output, we take the x-position of the maximum elevation and store it in an array. We can now plot the x position as a function of the time. There is some non-linear behaviour right in the beginning that we ignore. Taking the slope of the last 8 points we find the phase speed as the slope from a linear fit. The phase speed output from the code is thus $c = 3.24$ m/s. This corresponds to a 3.8% deviation from the theoretical value. This could be due to nummerical errors, or due to some simplifying assumptions made in the code. As a rule of thumb, the numerical solution should have a time step, dt, smaller than $\frac{dx}{c}$, to provide a stable solution. Smaller dt might provide a more accurate result, at the cost of computing time.

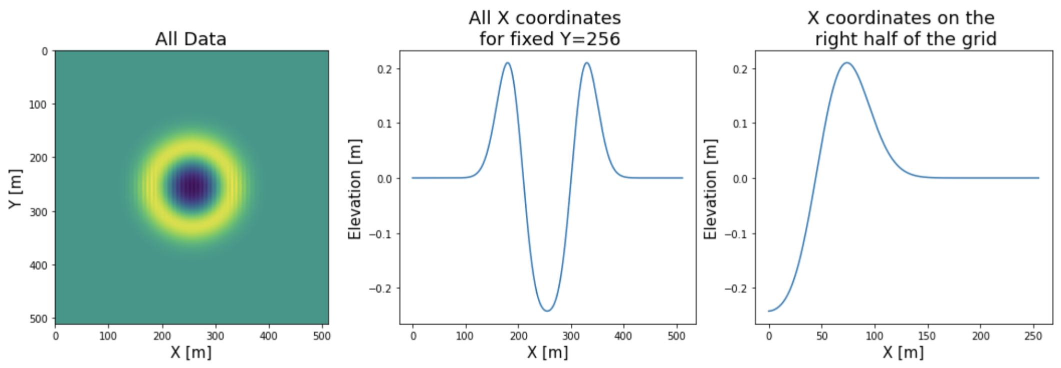

Figure 2: Shows how we start by having elevation for 512x512, this is reduced to elevation for 256 points for X, and we end up analysing only half of this as seen if panel 3.

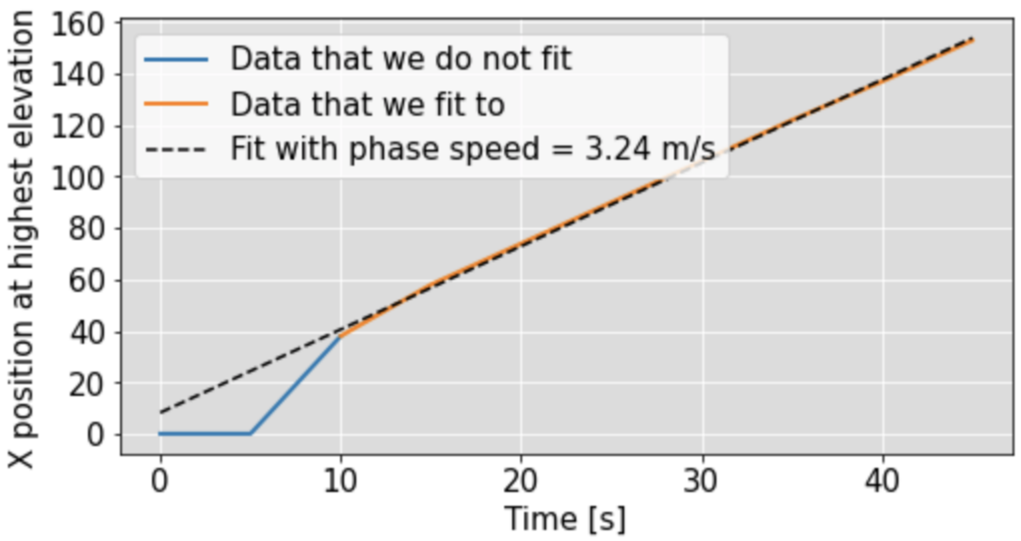

Figure 3: The x-position of the highest elevation as a function of time. A linear fit was made to the last 8 data points to determine the phase speed.

4 The Code

```
#include <vector>
#include <iostream>
#include <fstream>
#include <chrono>
#include <cmath>
#include <numeric>
#include <cassert>
#include <array>
#include <algorithm>

using real_t = float;
constexpr size_t NX = 2*512, NY = 2*512; //World Size
using grid_t = std::array<std::array<real_t, NX>, NY>;

class Sim_Configuration {
public:
    int iter = 1000; // Number of iterations
    double dt = 0.05; // Size of the integration time step
    real_t g = 9.80665; // Gravitational acceleration
    real_t dx = 1; // Integration step size in the horizontal direction
    real_t dy = 1; // Integration step size in the vertical direction
    int data_period = 100; // how often to save coordinate to file
    std::string filename = "sw_output.data"; // name of the output file with history

    Sim_Configuration(std::vector <std::string> argument){
        for (long unsigned int i = 1; i<argument.size() ; i += 2){
            std::string arg = argument[i];
            if(arg=="-h"){ // Write help
                std::cout << "./par␣--iter␣<number␣of␣iterations>␣--dt␣<time␣step>"
                          << "␣--g␣<gravitational␣const>␣--dx␣<x␣grid␣size>␣--dy␣<y␣grid␣size>"
                          << "--fperiod␣<iterations␣between␣each␣save>␣--out␣<name␣of␣output␣file>\n";
                exit(0);
            } else if (i == argument.size() - 1)
                throw std::invalid_argument("The␣last␣argument␣(" + arg +")␣must␣have␣a␣value");
            else if(arg=="--iter"){
                if ((iter = std::stoi(argument[i+1])) < 0)
                    throw std::invalid_argument("iter␣most␣be␣a␣positive␣integer␣(e.g.␣-iter␣1000)");
            } else if(arg=="--dt"){
                if ((dt = std::stod(argument[i+1])) < 0)
                    throw std::invalid_argument("dt␣most␣be␣a␣positive␣real␣number␣(e.g.␣-dt␣0.05)");
            } else if(arg=="--g"){
                g = std::stod(argument[i+1]);
            } else if(arg=="--dx"){
                if ((dx = std::stod(argument[i+1])) < 0)
                    throw std::invalid_argument("dx␣most␣be␣a␣positive␣real␣number␣(e.g.␣-dx␣1)");
            } else if(arg=="--dy"){
                if ((dy = std::stod(argument[i+1])) < 0)
                    throw std::invalid_argument("dy most be a positive real number (e.g. --dy 1)");
            } else if(arg=="--fperiod"){
                if ((data_period = std::stoi(argument[i+1])) < 0)
                    throw std::invalid_argument("dy most be a positive integer (e.g. --fperiod 100)");
            } else if(arg=="--out"){
                filename = argument[i+1];
            } else{
                std::cout << "---> error: the argument type is not recognized \n";
            }
        }
    }
};

/** Representation of a water world including ghost lines, which is a "1-cell padding" of rows and columns
 * around the world. These ghost lines is a technique to implement periodic boundary conditions. */
class Water {
public:
    grid_t u{}; // The speed in the horizontal direction.
    grid_t v{}; // The speed in the vertical direction.
    grid_t e{}; // The water elevation.
    Water() {
        for (size_t i = 1; i < NY - 1; ++i)
        for (size_t j = 1; j < NX - 1; ++j) {
            real_t ii = 100.0 * (i - (NY - 2.0) / 2.0) / NY;
            real_t jj = 100.0 * (j - (NX - 2.0) / 2.0) / NX;
            e[i][j] = std::exp(-0.02 * (ii * ii + jj * jj));
        }
    }
};

/* Write a history of the water heights to an ASCII file
 *
 * @param water_history Vector of the all water worlds to write
 * @param filename The output filename of the ASCII file
*/
void to_file(const std::vector<grid_t> &water_history, const std::string &filename){
    std::ofstream file(filename);
    file.write((const char*)(water_history.data()), sizeof(grid_t)*water_history.size());
}

/** Exchange the horizontal ghost lines i.e. copy the second data row to the very last data row and vice versa.
 *
 * @param data The data update, which could be the water elevation `e` or the speed in the horizontal direction `u`.
 * @param shape The shape of data including the ghost lines.
 */
void exchange_horizontal_ghost_lines(grid_t& data) {
    // Parallise this segment, let it know that "data" has already been copied
    #pragma acc parallel loop present(data)
    for (uint64_t j = 0; j < NX; ++j) {
            data[0][j] = data[NY−2][j];
            data[NY−1][j] = data[1][j];
        }
}

/** Exchange the vertical ghost lines i.e. copy the second data column to the rightmost data column and vice versa.
 *
 * @param data The data update, which could be the water elevation 'e' or the speed in the vertical direction 'v'.
 * @param shape The shape of data including the ghost lines.
 */
void exchange_vertical_ghost_lines(grid_t& data) {
    // Parallise this segment, let it know that "data" has already been copied
    #pragma acc parallel loop present(data)
    for (uint64_t i = 0; i < NY; ++i) {
        data[i][0] = data[i][NX−2];
        data[i][NX−1] = data[i][1];
    }
}

/** One integration step
 *
 * @param w The water world to update.
 */
void integrate(Water &w, const real_t dt, const real_t dx, const real_t dy, const real_t g) {
    exchange_horizontal_ghost_lines(w.e);
    exchange_horizontal_ghost_lines(w.v);
    exchange_vertical_ghost_lines(w.e);
    exchange_vertical_ghost_lines(w.u);

    // Rewrite the next two loops in to 1, parllise it and let it know which data it already has
    #pragma acc parallel loop collapse(2) present(w.u,w.v,w.e)
    for (uint64_t i = 0; i < NY − 1; ++i)
    for (uint64_t j = 0; j < NX − 1; ++j) {
        w.u[i][j] −= dt / dx * g * (w.e[i][j+1] − w.e[i][j]);
        w.v[i][j] −= dt / dy * g * (w.e[i + 1][j] − w.e[i][j]);
    }

    // Rewrite the next two loops in to 1, parllise it and let it know which data it already has
    #pragma acc parallel loop collapse(2) present(w.u,w.v,w.e)
    for (uint64_t i = 1; i < NY − 1; ++i)
    for (uint64_t j = 1; j < NX − 1; ++j) {
        w.e[i][j] −= dt / dx * (w.u[i][j] − w.u[i][j−1])
                   + dt / dy * (w.v[i][j] − w.v[i−1][j]);
    }
}

/** Simulation of shallow water
 *
 * @param num_of_iterations The number of time steps to simulate
 * @param size The size of the water world excluding ghost lines
 * @param output_filename The filename of the written water world history (HDF5 file)
 */
void simulate(const Sim_Configuration config) {
    Water water_world = Water();

    std::vector <grid_t> water_history;
    auto begin = std::chrono::steady_clock::now();

    // Copy the data from the CPU to the GPU's, so they have all available before starting their work
    #pragma acc data copy(water_world)
    for (uint64_t t = 0; t < config.iter; ++t) {
        integrate(water_world, config.dt, config.dx, config.dy, config.g);
        if (t % config.data_period == 0) {

            // Update the water_world on the CPU, exporting data from the GPUs, before we save
            #pragma acc update self(water_world.e)
            water_history.push_back(water_world.e);
        }
    }

    auto end = std::chrono::steady_clock::now();

    to_file(water_history, config.filename);

    std::cout << "checksum: " << std::accumulate(water_world.e.front().begin(), water_world.e.back().end(), 0.0) << std
        ↪ ::endl;
    std::cout << "elapsed_time: " << (end - begin).count() / 1000000000.0 << " sec" << std::endl;
}

/** Main function that parses the command line and start the simulation */
int main(int argc, char **argv) {
    auto config = Sim_Configuration({argv, argv+argc});
    simulate(config);
    return 0;
}
```