Assignment #5

Emma (JFV491), Michael (LVP115) and Tobias (CWR879)

March 2024

1 Parallelization strategy

Our road to parallelization was rather bumpy. Our initial plan was to:

1. Parallelize all non-sequential loops, and add simple clauses.
2. Optimize data locality.
3. Use the profiler to perform finishing steps, such as adding asynchronous behaviour.

What ended up happening was a lot of going back and forth between 2. and 3.. This resulted in three versions of our code, which will now be explained in more depth. v1 has parallel regions added to all non-sequential loops with #**pragma acc parallel for**, and relevant simple clauses. **num_gangs** was also added to all loops to test scaling. We also tried using #**pragma acc kernel**, but from the -Minfo output we saw that it did not manage to parallelize the code. We discovered that trying to parallelize the **Water** class causes errors, and since it is only called once we ignored it. Any attempts at parallelizing the two nested **for** loops in **integrate** did not work when we did not optimize data locality. Thus we do not expect any significant speedup for v1.

v2 added optimized data locality and simple asynchronous task scheduling. In **simulate**, the **integrate** function takes the same **water_history** for all time steps, so we added a **copyin(water_world)** outside the **simulate** loop. We needed to push updates to **water_history**, so we added #**pragma acc update self(water_world.e)** inside the **if** statement. Now we needed to tell everything inside the **integrate** function that **water_world** is present. This was done in the ghost line functions and the two nested loops in **integrate** with the **present(w)** clause added to the #**pragma acc parallel loop** directives. With the aid of profiling, we saw that our greatest bottleneck now was time spent on the PCI bus, as there was unnecessary travel between the GPU and CPU. To reduce this, we looked at making the ghost lines asynchronous. Of the four ghost line exchanges, only **exchange_horizontal_ghost_lines(w.e)** and **exchange_vertical_ghost_lines(w.e)** are dependent on each other. Thus, we can add a **asynch** clause to the ghost line **parallel loop** directives, and a #**pragma acc wait** after exchanging **w.v**, **w.u** and either of the two **w.e** exchanges, followed by the remaining **w.e** ghost line exchange. Then we need another **wait** as the nested loops below depend on **w.e**.

The final optimized version, v3, added proper asynchronous behaviour using queues. The **async** clause was added to loop directives inside the ghost line functions, and the functions were given a queue number as a variable. The two **w.e** ghost lines were added to the same queue, while **w.v** and **w.u** had their own queues. All ghost line exchanges must be finished before moving on to the loops, which was achieved by adding two **wait** statements, one for **w.v** and one for **w.u**. We could add a **wait** statement for **w.e** as well, but it is more optimal to have the two existing **wait** directives be done asynchronously in the same queue as **w.e**. Then, if we also do the two nested **for** loops asynchronously in **w.e**'s queue, we achieve the goal of having all ghost line exchanges finished before the nested loops, but with the speedup **async** gives by reducing the update signals between the CPU and GPU. After implementing this, using the profiler showed that the GPU was active for almost 100% of the time it took to run the code, which is the ideal case.

We looked at other minor things, such as using **tile** for the nested **for** loops. In this case, since the loops only go through a minor part of the grid at each iteration (as compared to e.g. transposing a matrix which accesses full rows and columns), we found that **collapse** worked better.

2 Scaling

The relative strong and weak scaling can be seen in fig 1. We scaled the problem by a factor of 16 for the strong scaling. We tested the scaling for a number of gangs equal to $1, 2, 4, 8, 14$, with 14 being the maximum number of SMs available. From fitting, we found the parallelization of the programs to be:

$$\begin{aligned}
P^{v1}_{\text{strong}} &\approx 0. \\
P^{v2}_{\text{strong}} &= 0.975\pm, 0.004 \\
P^{v3}_{\text{strong}} &= 0.965\pm, 0.001 \\
P^{v1}_{\text{weak}} &= 0.05\pm, 0.01 \\
P^{v2}_{\text{weak}} &= 0.27\pm, 0.03 \\
P^{v3}_{\text{weak}} &= 0.29\pm, 0.03
\end{aligned}$$

Now the parallelization fraction between strong and weak scaling does not agree all too well. Also, it is worth mentioning that while the relative performance of v3 may not be much better than v2, the absolute performance was superior (v3 was roughly twice as fast as v2 for 14 gangs with strong scaling), which profiling also confirmed. This might have been better reflected in our above results if we had chosen a bigger problem size.

For our v1 program, we ran into a weird issue when scaling the problem by increasing N_x and N_y, so instead we scaled it using the number of time iterations **iter**. We also noticed that performance and uncertainty were heavily dependent on the current Erda user number (i.e. bad performance during exercise classes, excellent performance late Friday night)

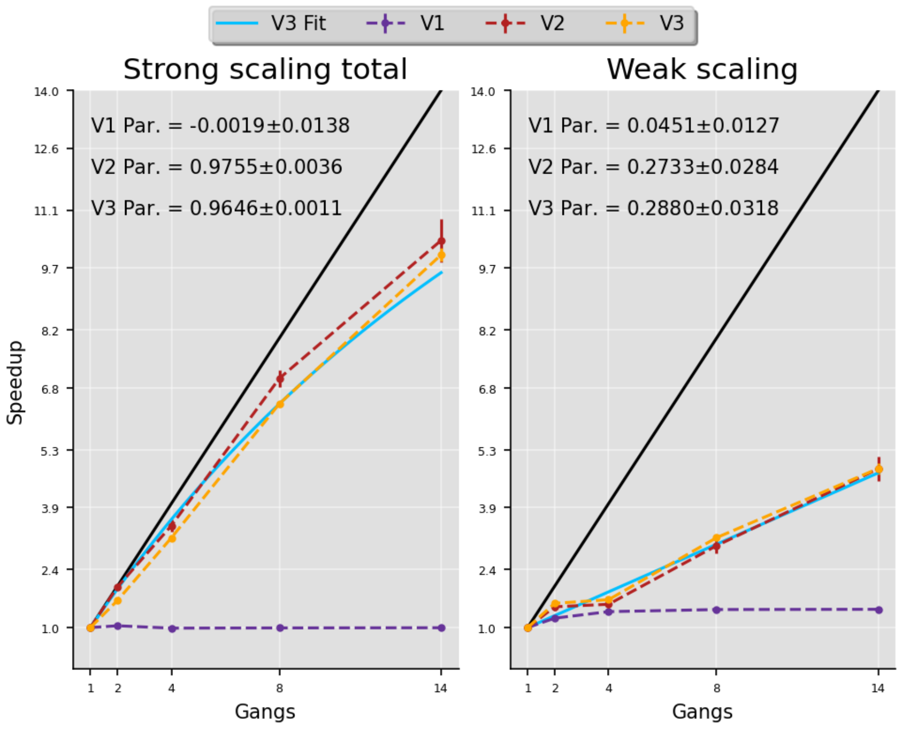

Figure 1: Strong (left) and weak (right) relative scaling for the three versions of our code together with the fit for v3.

C++ Code

2.1 v1

```
#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 = 512, NY = 512; //World Size
using grid_t = std::array<std::array<real_t , NX>, NY>;
#define NUM_GANGS 200

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 h

    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
                          << "—fperiod␣<iterations␣between␣each␣save>␣—out␣<na
                exit(0);
            } else if (i == argument.size() − 1)
                throw std::invalid_argument("The␣last␣argument␣(" + arg +")␣must
            else if(arg=="—iter"){
                if ((iter = std::stoi(argument[i+1])) < 0)
                    throw std::invalid_argument("iter␣most␣be␣a␣positive␣integer
            } else if(arg=="—dt"){
                if ((dt = std::stod(argument[i+1])) < 0)
                    throw std::invalid_argument("dt␣most␣be␣a␣positive␣real␣numb
            } 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␣numb
            } else if(arg=="--dy"){
                if ((dy = std::stod(argument[i+1])) < 0)
                    throw std::invalid_argument("dy␣most␣be␣a␣positive␣real␣numb
            } else if(arg=="--fperiod"){
                if ((data_period = std::stoi(argument[i+1])) < 0)
                    throw std::invalid_argument("dy␣most␣be␣a␣positive␣integer␣(
            } 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 pa
 *  around the world. These ghost lines is a technique to implement periodic bou
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 &filenam
    std::ofstream file(filename);
    file.write((const char*)(water_history.data()), sizeof(grid_t)*water_history
}
```

```
/** Exchange the horizontal ghost lines i.e. copy the second data row to the ver
 *
 * @param data   The data update, which could be the water elevation `e` or the
 * @param shape  The shape of data including the ghost lines.
 */
void exchange_horizontal_ghost_lines(grid_t& data) {
    #pragma acc parallel loop num_gangs(NUM_GANGS)
    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 ri
 *
 * @param data   The data update, which could be the water elevation `e` or the
 * @param shape  The shape of data including the ghost lines.
 */
void exchange_vertical_ghost_lines(grid_t& data) {
    #pragma acc parallel loop num_gangs(NUM_GANGS)
    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, cons
        exchange_horizontal_ghost_lines(w.v);
        exchange_horizontal_ghost_lines(w.e);
        exchange_vertical_ghost_lines(w.u);
        exchange_vertical_ghost_lines(w.e);

        // #pragma acc parallel loop collapse(2) num_gangs(NUM_GANGS)
        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]);
        }

        // #pragma acc parallel loop collapse(2) num_gangs(NUM_GANGS)
        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 (HD
 */
void simulate(const Sim_Configuration config) {
    Water water_world = Water();

    std::vector <grid_t> water_history;
    auto begin = std::chrono::steady_clock::now();
    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) {
            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(),
    std::cout << "elapsed␣time:␣" << (end - begin).count() / 1000000000.0 << "␣s
}

/** 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;
}
```

2.2 v2

```
#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 = 512, NY = 512; //World Size
using grid_t = std::array<std::array<real_t , NX>, NY>;
#define NUM_GANGS 4

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 h

    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
                          << "—fperiod␣<iterations␣between␣each␣save>␣—out␣<na
                exit(0);
            } else if (i == argument.size() - 1)
                throw std::invalid_argument("The␣last␣argument␣(" + arg +")␣must
            else if(arg=="—iter"){
                if ((iter = std::stoi(argument[i+1])) < 0)
                    throw std::invalid_argument("iter␣most␣be␣a␣positive␣integer
            } else if(arg=="—dt"){
                if ((dt = std::stod(argument[i+1])) < 0)
                    throw std::invalid_argument("dt␣most␣be␣a␣positive␣real␣numb
            } 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 numb
            } else if(arg=="--dy"){
                if ((dy = std::stod(argument[i+1])) < 0)
                    throw std::invalid_argument("dy most be a positive real numb
            } else if(arg=="--fperiod"){
                if ((data_period = std::stoi(argument[i+1])) < 0)
                    throw std::invalid_argument("dy most be a positive integer (
            } 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 pa
 *  around the world. These ghost lines is a technique to implement periodic bou
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() {
        // #pragma acc parallel loop gang vector collapse(2) copy(e)
        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 &filenam
    std::ofstream file(filename);
    file.write((const char*)(water_history.data()), sizeof(grid_t)*water_history
}

/** Exchange the horizontal ghost lines i.e. copy the second data row to the ver
```

```
 *
 * @param data   The data update, which could be the water elevation `e` or the
 * @param shape  The shape of data including the ghost lines.
 */
void exchange_horizontal_ghost_lines(grid_t& data) {
    #pragma acc parallel loop async present(data) num_gangs(NUM_GANGS)
    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 rig
 *
 * @param data   The data update, which could be the water elevation `e` or the
 * @param shape  The shape of data including the ghost lines.
 */
void exchange_vertical_ghost_lines(grid_t& data) {
    // #pragma acc data present(data)
    #pragma acc parallel loop async present(data) num_gangs(NUM_GANGS)
    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, cons
    // Problems:
    // 1. present(w) - Might be solved? Is it present in the ghost lines?
        // Want to make water_world w present to the expresssions inside the int
        // Not sure if should do it for all expressions simultaneously, or indiv
    // 2. Parallization not applied. Run in sequential.
        // Want to make a single parallel region for the integrate function
        // Ghost lines are run in sequential
    // #pragma acc parallel num_gangs(NUM_GANGS) present(w)
// Give shape of w? But w is class
        exchange_horizontal_ghost_lines(w.v);
        exchange_horizontal_ghost_lines(w.e);
        exchange_vertical_ghost_lines(w.u);
        #pragma acc wait  // Need to wait as the next line is dependent on the p
        exchange_vertical_ghost_lines(w.e);
        #pragma acc parallel loop present(w) collapse(2) num_gangs(NUM_GANGS) //
        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]);
        }

        #pragma acc parallel loop present(w) collapse(2) num_gangs(NUM_GANGS)
        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 (HD
 */
void simulate(const Sim_Configuration config) {
    Water water_world = Water();

    std::vector <grid_t> water_history;
    auto begin = std::chrono::steady_clock::now();
    #pragma acc data copyin(water_world)
    for (uint64_t t = 0; t < config.iter; ++t) {  // K res seq
        integrate(water_world, config.dt,  config.dx, config.dy, config.g);
        if (t % config.data_period == 0) {
            #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(),
    std::cout << "elapsed time: " << (end - begin).count() / 1000000000.0 << " s
}

/** 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;
```

}

2.3 v3

```
#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 = 512*4, NY = 512*4; //World Size
using grid_t = std::array<std::array<real_t , NX>, NY>;
#define NUM_GANGS 200

class Sim_Configuration {
public:
    // OBS skal vi   ndre   de her?
    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 h

    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␣ste
                          << "␣--g␣<gravitational␣const>␣--dx␣<x␣grid␣size>␣--dy
                          << "--fperiod␣<iterations␣between␣each␣save>␣--out␣<na
                exit(0);
            } else if (i == argument.size() - 1)
                throw std::invalid_argument("The␣last␣argument␣(" + arg +")␣must
            else if(arg=="--iter"){
                if ((iter = std::stoi(argument[i+1])) < 0)
                    throw std::invalid_argument("iter␣most␣be␣a␣positive␣integer
            } else if(arg=="--dt"){
                if ((dt = std::stod(argument[i+1])) < 0)
                    throw std::invalid_argument("dt␣most␣be␣a␣positive␣real␣numb
            } 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 numb
            } else if(arg=="--dy"){
              if ((dy = std::stod(argument[i+1])) < 0)
                throw std::invalid_argument("dy most be a positive real numb
            } else if(arg=="--fperiod"){
              if ((data_period = std::stoi(argument[i+1])) < 0)
                throw std::invalid_argument("dy most be a positive integer (
            } 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 pa
 *  around the world. These ghost lines is a technique to implement periodic bou
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() {
        // #pragma acc parallel loop gang vector collapse(2) copy(e)
// Often causes errors, commented out
        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 &filenam
    std::ofstream file(filename);
    file.write((const char*)(water_history.data()), sizeof(grid_t)*water_history
}
```

```
/** Exchange the horizontal ghost lines i.e. copy the second data row to the ver
 *
 * @param data   The data update, which could be the water elevation `e` or the
 * @param shape  The shape of data including the ghost lines.
 */
void exchange_horizontal_ghost_lines(grid_t& data, int qnum) {
    #pragma acc parallel loop async present(data) num_gangs(NUM_GANGS)
// OBS Skal man sige hvad st rrelse data har? Fx data[1:]
    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 ri
 *
 * @param data   The data update, which could be the water elevation `e` or the
 * @param shape  The shape of data including the ghost lines.
 */
void exchange_vertical_ghost_lines(grid_t& data, int qnum) {
    #pragma acc parallel loop async(qnum) present(data) num_gangs(NUM_GANGS)
    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, cons
        exchange_horizontal_ghost_lines(w.v, 2);
        exchange_horizontal_ghost_lines(w.e, 1);
        exchange_vertical_ghost_lines(w.u, 4);
        exchange_vertical_ghost_lines(w.e, 1);
        #pragma acc wait(2) async(1)
        #pragma acc wait(4) async(1)

        #pragma acc parallel loop async(1) present(w) collapse(2) num_gangs(NUM_
        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]);
        }

        #pragma acc parallel loop async(1) present(w) collapse(2) num_gangs(NUM_
        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 (HD
 */
void simulate(const Sim_Configuration config) {
    Water water_world = Water();

    std::vector <grid_t> water_history;
    auto begin = std::chrono::steady_clock::now();
    #pragma acc data copyin(water_world)
    for (uint64_t t = 0; t < config.iter; ++t) {  // K res seq
        integrate(water_world, config.dt,  config.dx, config.dy, config.g);
        if (t % config.data_period == 0) {
            #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(),
    std::cout << "elapsed time: " << (end - begin).count() / 1000000000.0 << " s
}

/** 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;
}
```