Attempt 1 - Parallelization in one dimension

After having been acquainted with the code and, I started off with sketching the problem on pen and paper. I found that most natural way to parallelize the problem was to utilize the current functions to the extent possible, and this meant dividing the global domain into local ones by creating both the global and local domains using the createWater function, but with different size arguments. In both cases the domains are created with the ghostlines. To avoid unnecessary communication, the global domain is created in each process.

"" // Parallellize ONLY in one dimension. Columns stay the same, rows are divided auto rows_local = size/numprocs;

// Create global world Shape shape_with_ghost_lines = Shape(size + 2, size + 2);
Water water_world = createWater(shape_with_ghost_lines, Shape(size, size), -1, -1);

// Create local world Shape shape_with_ghost_lines_local = Shape(rows_local + 2, size + 2);
Water world_local = createWater(shape_with_ghost_lines_local, Shape(rows_local, size), 0, 0);"

After this, the initialized global domain (with a droplet in the middle) needs to be divided between the processes (the local u is set to a subset of the global u where the position depends on the processor rank).

Finally for the main function, the local water elevations (after removing the ghostlines) need to be gathered on the main process:

"" std::vector e_glob(size*size);
std::vector e_loc = remove_ghost_lines(world_local.e, shape_with_ghost_lines_local, true, true);

// gather the water elevation e on rank zero
MPI_Gather(&e_loc[0], rows_local.size, MPI_DOUBLE, &e_glob[0], rows_local.size, MPI_DOUBLE, ROOT, MPI_COMM_WORLD);"

The actual communication between processors, where the ghostlines are exchanged before each integration steps, was implemented relatively easily in the 1D case, the reason being that the arrays remain contiguous in memory. The function for exchanging horizontal ghost lines was changed into an mpi version which contains only two lines for the communication:

MPI_Sendrecv(&data[top_water], shape.cols, MPI_DOUBLE, NORTH, 123, &data[bot_ghost], shape.cols, MPI_DOUBLE, SOUTH, 123, MPI_COMM_WORLD);""
Attempt 1 - Parallelization in one dimension, non-blocking communication

In the previous example, the retrieval of ghostlines from the other processors (the northern and southern neighbor) was with a blocking communication, before any computations. With this problem, it is entirely possible to do computations while waiting for the communication to finish, since only the first and last rows (and columns in the 2D case) need information from neighbouring processes. The shallow water equation can be solved for all the “inner” grid cells independently of other processes. Thus, overlapping computations with communication can be done by

1. starting a non-blocking communication of the ghostlines with north and south neighbours (using Irecv and Isend) for \( v \) and \( e \)
2. Doing calculations for the inner grid, in the 1D case, the inner rows (first and last local columns can still be calculated)
3. Wait for the communication to finish by MPI_Waitall
4. Calculate the first and last rows of the local domain

The fact that the ghostlines are exchanged inside a function means the MPI-Request object needs to be passed between functions, e.g. created first inside the integrate function, passed to exchange_horizontal_code_lines_mpi, and and passed back to the computational function to be used with MPI_Waitall. Unfortunately, I couldn’t work out how to do this in C++ since this is a new language to me, even after several hours of trying. In the end, I had to move the ghost line communication inside the integrate function, which makes the code a lot uglier but works.

With latency hiding, the runtime went from 4.3 to 3.4 seconds on DAG using 4 processors, –size 2000 and –iter 400.

Attempt 2 - Parallelization in both dimensions

For parallelization in both dimensions using MPI (the 2D grid is divided into both subcolumns and subrows) the issue is complicated by memory not being contiguous anymore, which means that e.g. when communicating the outer columns, the data cannot be sent using a normal MPI_Isend command.

For the 2D case, we could utilize a Cartesian topology which is a built-in functionality in MPI. This has at least two potential benefits: 1) getting the functions N/S/W/E neighbours is automatic, and 2) it gives MPI the chance for some optimizations regarding which processes are assigned as neighbours. While it’s true that 1) is trivial to implement manually and 2) is unlikely to lead to tangible improvements in DAG as this is presumably a homogenous cluster where all the computers can talk to each other with similar latency, I decided to nevertheless get acquainted with the Cartesian communicator in MPI.

The initialization of local worlds remains fairly trivial:

```c
ierr = MPI_Cart_create(old_comm, ndims, dim_size, wrap_around, reorder, &comm2d);
```
if(ierr != 0) printf("ERROR[%d] creating CART\n", ierr);

// find my coordinates in the cartesian communicator group
MPI_Cart_coords(comm2d, mpi_rank, ndims, coordinates);
printf("I used to be [%d] now I am [%d], [%d]\n", mpi_rank, coordinates[0], coordinates[1]);

// Parallelize in two dimensions
// Number of rows and columns in a local world
auto nrows_local = size/(numprocs/2);
auto ncols_local = size/(numprocs/2);

printf("size is: [%d]*[%d], local size is: [%d]*[%d] \n", size, size, nrows_local, ncols_local);

// Create global world
Shape shape_with_ghost_lines = Shape(size + 2, size + 2);
Water water_world = createWater(shape_with_ghost_lines, Shape(size, size), -1, -1);

// Create local world
Shape shape_with_ghost_lines_local = Shape(nrows_local + 2, ncols_local + 2);
Water world_local = createWater(shape_with_ghost_lines_local, Shape(nrows_local, ncols_local), 0, 0);

//uint64_t offset_cols = mpi_rank * rows_local * world_local.shape.cols;
uint64_t i_global, j_global, iabs1, iabs2;
for (uint64_t i = 1; i < world_local.shape.rows - 1; ++i) {
    i_global = i + ((coordinates[0]) * nrows_local);
    for (uint64_t j = 1; j < world_local.shape.cols - 1; ++j) {
        j_global = j + ((coordinates[1]) * ncols_local);

        world_local.e[i * world_local.shape.cols + j] = water_world.e[i_global * water_world.shape.cols + j_global];
        printf("i_local: [%d], j_local: [%d], i_global: [%d], j_global: [%d] \n", i, j, i_global, j_global);
    }
}

I added some print functions to make sure that the array was initialized correctly.

Unfortunately, upon inspecting the gifs I realized that something had gone wrong. Despite the local worlds being initialized correctly, the gather operation on rank 0 had not collected the local worlds correctly. The fact that incoming local arrays need to be arranged non-contiguously in the global array proved to be extremely difficult to solve. I tried to solve this problem using custom MPI_Types (MPI_Type_create_subarray) and the Gatherv function for displacements of the small arrays into the large arrays memory. However, in the end I could not make it work. This was frustrating as the ghostline communication itself should have been fairly easy once everything was setup. Moreover, it is not recommended to “gather” local operations on a single process (rank 0) when it’s possible to do the data saving itself in parallel using MPI-IO, but I assumed the gather operation was supposed to be used in this assignment.

The improvements in running time based on running the 1D parallelized code on MODI is shown in Figure 1. The results are fairly disappointing with weak scaling. Despite the frustrations with C++, and the failure to make 2D parallelization work due to stumbling on the “Gather” operation, this was a great learning experience!
Figure 1: Parallelization of shallow water model