Assignment #6

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

March 2024

1 MPI Parallelization strategy

Our overall intended strategy was as follows:

1. Decompose the domain into one-dimensional slabs.
2. Reduce checksum and stats to rank 0, and only have that processor print it.
3. Gather all the different sized ranks' data to rank 0 and write only from rank 0 to the output file (i.e. I/O).
4. Use Two-dimensional decomposition.

As it turned out, this was unfortunately not an ideal strategy and because of time constraints, we did not manage to do all of the above. The better strategy would have been to first decompose the domain and then fix the I/O, not the other way around as we tried. The I/O is much easier for a 2d decomposition than a 1d vertical slab decomposition.
First, we decomposed the problem into a vertical slap geometry and then updated the ghost cells accordingly. The ghost cell updates were done by sending the second and second-to-last columns to the left and right ranks respectively, taking periodicity into account. The columns were sent and received nonblockingly. Then we moved on to the **checksum** and **stat** functions and made sure all the local world data was collected on rank 0 and only printed by that rank. In **checksum** this was done using **MPI_Reduce** with the **MPI_SUM** argument. In **stat** we similarly reduced the minimum, maximum and mean temperature, using **MPI_MIN**, **MPI_MAX** and **MPI_SUM** respectively. We could use **MPI_SUM** for the mean temperature since the local means were calculated using the global grid size. Unfortunately, this was as far as time allowed us to progress. We made progress on the two-dimensional decomposition and I/O gathering, which will be described now. We tried manually (as opposed to using the inbuild MPI cartesian geometry and shift functions) two-dimensionalizing by repeating what we did with the vertical slabs, but horizontally. The only real difference is that the neighbouring ranks were not as simple as the current rank ± 1. We did not figure out how to get the ranks of the domains above and

below the current rank.
In the I/O, to collect all ranks' data into the global world data, we first removed the ghost cells, and then we gathered all the data. The ranks do not have the same dimensions, which means we had to use **Gatherv** which takes different offsets and data sizes. After gathering the data, we now ran into an issue with reordering it, and because we used vertical slabs this proved too difficult. We tried switching to horizontal slabs, which would *not* have needed any reordering but again we ran out of time. In hindsight, it would have been better to get the 2d domain decomposition to work before working on I/O.

2 Scaling

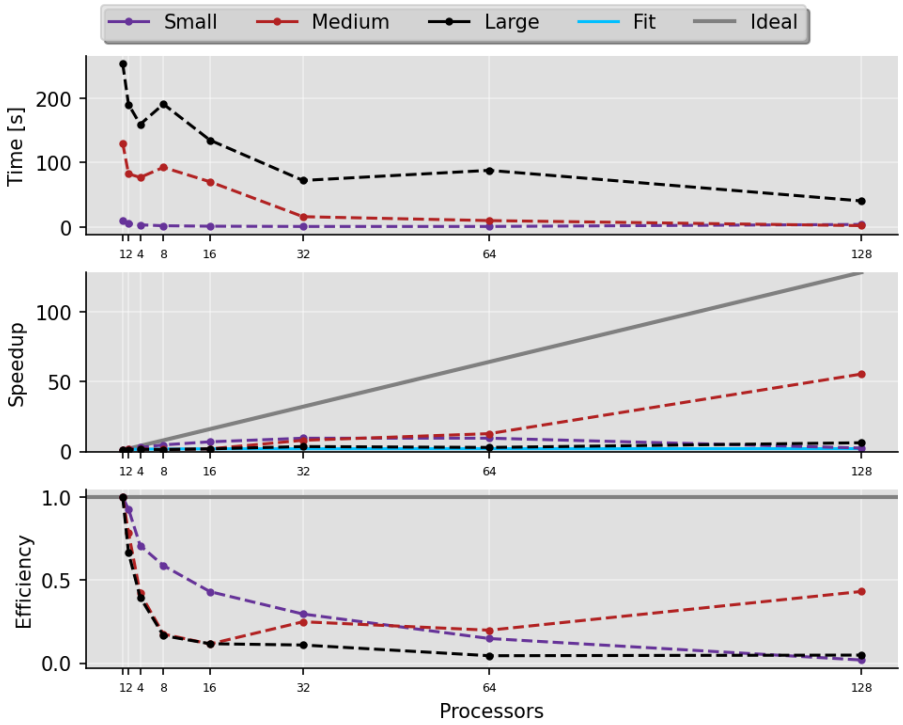

Figure 1: Strong scaling shown as absolute, relative and efficiency for the small, medium and large model. Each relative scaling graph was fitted to Ahmdahl's law, but only the large model fit is shown. Uncertainties are often smaller than the points

In fig 1, our results can be seen. The larger models' runtime became several minutes, so we had to adjust the iterations. We used 10000 iterations for the small model, 1000 for the medium and 500 for the large. For an unknown

reason, when using 2 nodes in the small model the checksum changed, so take that point with a grain of salt. The absolute scaling graph has a few "bumps", which are due to ERDA's memory architecture. At $N_{core} = 8$ and 64 we see two visible bumps. The first at 8 cores is because ERDA's architecture places 4 processors close to each other, so when going from 4 to 8, we go from one of these sets of 4 close processor groups to two groups, which increases the length data has to travel and thus introduces a latency overhead. While not nearly as visible as the earlier bump, going from 64 to 128 processors means increasing the number of nodes from 1 to 2, again increasing the travel distance and thus latency. For the larger workload, we should expect our parallelized version to perform relatively better. From the graphs in fig 1, this seems to be mostly true for the medium model, but this could be because we decreased the iterations on the large model. After all, as seen in the absolute scaling graph, it took around three minutes to run the large model for a low number of cores, and we prioritized repeated measurements instead of more iterations. The parallel fractions, found by fitting the relative speedup data, are:

$$P_{\text{small}} = 0.907 \pm 0.009 \tag{1}$$

$$P_{\text{medium}} = 0.47 \pm 0.09 \tag{2}$$

$$P_{\text{large}} = 0.50 \pm 0.08 \tag{3}$$

We took three measurements per core for the small model, and only two measurements for the medium and large models, again, because of the large computation time. This means the uncertainty on the points is small, which affects the fits, and the uncertainty of the parallelization values would most likely be higher had we taken more points. Otherwise, the results suggest well-parallelized code. The lower fraction on the medium and large models could be because of the lower number of iterations.

C++ Code

```
#include <vector>
#include <iostream>
#include <H5Cpp.h>
#include <chrono>
#include <cmath>
#include <numeric>
#include <mpi.h>

// Get the number of processes
int mpi_size;

// Get the rank of the process
int mpi_rank;

/** Representation of a flat world */
class World {
public:
    // The current world time of the world
    double time;
    // The size of the world in the latitude dimension and the global size
    uint64_t latitude, global_latitude;
    // The size of the world in the longitude dimension
    uint64_t longitude, global_longitude;
    // The offset for this rank in the latitude dimension
    long int offset_latitude;  // Local cell to global
    // The offset for this rank in the longitude dimension
    long int offset_longitude;
    // The temperature of each coordinate of the world.
    // NB: it is up to the calculation to interpret this vector in two dimension
    std::vector<double> data;
    // The measure of the diffuse reflection of solar radiation at each world co
    // See: <https://en.wikipedia.org/wiki/Albedo>
    // NB: this vector has the same length as `data` and must be interpreted in
    std::vector<double> albedo_data;

    /** Create a new flat world.
     *
     * @param latitude     The size of the world in the latitude dimension.
     * @param longitude    The size of the world in the longitude dimension.
     * @param temperature  The initial temperature (the whole world starts with
     * @param albedo_data  The measure of the diffuse reflection of solar radiat
     *                     This vector must have the size: `latitude * longitude
     */
```

```
    World(uint64_t latitude, uint64_t longitude, double temperature,
          std::vector<double> albedo_data) : latitude(latitude), longitude(longi
                                             data(latitude * longitude, temperat
                                             albedo_data(std::move(albedo_data))
};

double checksum(World &world) {
    // Every rank loops through it's elements, and calculates cs.
    // Then send all cs values to one rank and sum them up, using reduce.


    // TODO: make sure checksum is computed globally
    // only loop *inside* data region -- not in ghostzones!
    double cs=0;
    for (uint64_t i = 1; i < world.latitude - 1; ++i)
    for (uint64_t j = 1; j < world.longitude - 1; ++j) {
        cs += world.data[i*world.longitude + j];
    }

    // Reduce all values
    double cs_combined;
    int send_count = 1;
    int receiving_rank = 0;
    MPI_Reduce(&cs, &cs_combined, send_count, MPI_DOUBLE, MPI_SUM, receiving_rank


    if (mpi_rank == 0){
        std::cout << "checksum------:" << cs_combined << std::endl;
    }

    return cs_combined;
}

void stat(World &world) {
    // TODO: make sure stats are computed globally

    // Reduce, Reduction, Allredcution to get global
    double mint = 1e99;
    double maxt = 0;
    double meant = 0;
    for (uint64_t i = 1; i < world.latitude - 1; ++i)
    for (uint64_t j = 1; j < world.longitude - 1; ++j) {
        mint = std::min(mint,world.data[i*world.longitude + j]);
        maxt = std::max(maxt,world.data[i*world.longitude + j]);
        meant += world.data[i*world.longitude + j];
    }
```

```
    meant = meant / (world.global_latitude * world.global_longitude);

    // Local to global
    int receiveing_rank = 0;

    // Get min temperature
    double mint_global;
    MPI_Reduce(&mint, &mint_global, 1, MPI_DOUBLE, MPI_MIN, receiveing_rank, MPI

    // Get max temperature
    double maxt_global;
    MPI_Reduce(&maxt, &maxt_global, 1, MPI_DOUBLE, MPI_MAX, receiveing_rank, MPI

    // Get mean temperature by summing up each rank's mean and then dividing by
    double meant_global;
    MPI_Reduce(&meant, &meant_global, 1, MPI_DOUBLE, MPI_SUM, receiveing_rank, M

    if (mpi_rank == 0){
    std::cout <<   "min:" << mint_global
              << ",max:" << maxt_global
              << ",avg:" << meant_global << std::endl;
    }
}

/** Exchange the ghost cells i.e. copy the second data row and column to the ver
 *
 * @param world  The world to fix the boundaries for.
 */
void exchange_ghost_cells(World &world) {
    // Probably finished TODO: figure out exchange of ghost cells bewteen ranks
    // Columns
        // For the current rank, send idx=1 column to rank-1, the idx=-2 column
        std::vector<double> column_send_left;  // Send to rank - 1
        std::vector<double> column_send_right;  // Send to rank + 1

        for (uint64_t i = 0; i < world.latitude; i++){
                column_send_left.push_back( world.data[i * world.longitude + 1])
                column_send_right.push_back(world.data[i * world.longitude + (wo
        }

        int left_rank = (mpi_rank + mpi_size - 1) % mpi_size;
// Source of Error regarding last rank?
        int right_rank = (mpi_rank + 1) % mpi_size;  // Source of Error regardin

        MPI_Request request[4];
```

```
        // Send to left and right
        double* pointer_to_send_left = &column_send_left[0];
        double* pointer_to_send_right = &column_send_right[0];
        MPI_Isend(pointer_to_send_left, world.latitude, MPI_DOUBLE, left_rank, 1
        MPI_Isend(pointer_to_send_right, world.latitude, MPI_DOUBLE, right_rank,

        // Receive left and right
        std::vector<double> column_left_ghost(world.latitude);
// The left column ghost cells
        std::vector<double> column_right_ghost(world.latitude);

        double* pointer_to_recv_left = &column_left_ghost[0];
        double* pointer_to_recv_right = &column_right_ghost[0];
        MPI_Irecv(pointer_to_recv_left, world.latitude, MPI_DOUBLE, left_rank, 2
        MPI_Irecv(pointer_to_recv_right, world.latitude, MPI_DOUBLE, right_rank,

        MPI_Waitall(4, request, MPI_STATUSES_IGNORE);  // OBS maybe request need

    // Apply Ghost cell update to local
    for (uint64_t i = 0; i < world.latitude; ++i) {
        world.data[i*world.longitude + 0] = column_left_ghost[i];
        world.data[i*world.longitude + world.longitude-1] = column_right_ghost[i
    }

        // Rows - For 1d slabs use internal.
    for (uint64_t j = 0; j < world.longitude; ++j) {
        world.data[0*world.longitude + j] = world.data[(world.latitude-2)*world.
        world.data[(world.latitude-1)*world.longitude + j] = world.data[1*world.
    }
}

/** Warm the world based on the position of the sun.
 *
 * @param world      The world to warm.
 */
void radiation(World& world) {
    double sun_angle = std::cos(world.time);
    double sun_intensity = 865.0;
    double sun_long = (std::sin(sun_angle) * (world.global_longitude / 2))
                      + world.global_longitude / 2.;
    double sun_lat = world.global_latitude / 2.;
    double sun_height = 100. + std::cos(sun_angle) * 100.;
    double sun_height_squared = sun_height * sun_height;

    for (uint64_t i = 1; i < world.latitude-1; ++i) {
        for (uint64_t j = 1; j < world.longitude-1; ++j) {
```

```
            // Euclidean distance between the sun and each earth coordinate
            double delta_lat  = sun_lat  - (i + world.offset_latitude);
            double delta_long = sun_long - (j + world.offset_longitude);
            double dist = sqrt(delta_lat*delta_lat +
                               delta_long*delta_long +
                               sun_height_squared);
            world.data[i * world.longitude + j] += \
                (sun_intensity / dist) * (1. - world.albedo_data[i * world.longi
        }
    }
    exchange_ghost_cells(world);
}

/** Heat radiated to space
 *
 * @param world  The world to update.
 */
void energy_emmision(World& world) {
    for (uint64_t i = 0; i < world.latitude * world.longitude; ++i) {
        world.data[i] *= 0.99;
    }
}

/** Heat diffusion
 *
 * @param world  The world to update.
 */
void diffuse(World& world) {
    std::vector<double> tmp = world.data;
    for (uint64_t k = 0; k < 10; ++k) {
        for (uint64_t i = 1; i < world.latitude - 1; ++i) {
            for (uint64_t j = 1; j < world.longitude - 1; ++j) {
                // 5 point stencil
                double center = world.data[i * world.longitude + j];
                double left = world.data[(i - 1) * world.longitude + j];
                double right = world.data[(i + 1) * world.longitude + j];
                double up = world.data[i * world.longitude + (j - 1)];
                double down = world.data[i * world.longitude + (j + 1)];
                tmp[i * world.longitude + j] = (center + left + right + up + dow
            }
        }
        std::swap(world.data, tmp);  // swap pointers for the two arrays
        exchange_ghost_cells(world); // update ghost zones
    }
}
```

```
/** One integration step at `world_time`
 *
 * @param world      The world to update.
 */
void integrate(World& world) {
    radiation(world);
    energy_emmision(world);
    diffuse(world);
}

/** Read a world model from a HDF5 file
 *
 * @param filename The path to the HDF5 file.
 * @return         A new world based on the HDF5 file.
 */
World read_world_model(const std::string& filename) {
    H5::H5File file(filename, H5F_ACC_RDONLY);
    H5::DataSet dataset = file.openDataSet("world");
    H5::DataSpace dataspace = dataset.getSpace();

    if (dataspace.getSimpleExtentNdims() != 2) {
        throw std::invalid_argument("Error while reading the model: the number o
    }

    if (dataset.getTypeClass() != H5T_FLOAT or dataset.getFloatType().getSize()
        throw std::invalid_argument("Error while reading the model: wrong data t
    }

    hsize_t dims[2];
    dataspace.getSimpleExtentDims(dims, NULL);
    std::vector<double> data_out(dims[0] * dims[1]);
    dataset.read(data_out.data(), H5::PredType::NATIVE_DOUBLE, dataspace, datasp
    std::cout << "World model loaded -- latitude: " << (unsigned long) (dims[0])
              << (unsigned long) (dims[1]) << std::endl;
    return World(static_cast<uint64_t>(dims[0]), static_cast<uint64_t>(dims[1]),
}

/** Write data to a hdf5 file
 *
 * @param group  The hdf5 group to write in
 * @param name   The name of the data
 * @param shape  The shape of the data
 * @param data   The data
 */
void write_hdf5_data(H5::Group& group, const std::string& name,
                     const std::vector<hsize_t>& shape, const std::vector<doubl
```

```
    H5::DataSpace dataspace(static_cast<int>(shape.size()), &shape[0]);
    H5::DataSet dataset = group.createDataSet(name.c_str(), H5::PredType::NATIVE
    dataset.write(&data[0], H5::PredType::NATIVE_DOUBLE);
}

/** Write a history of the world temperatures to a HDF5 file
 *s
 * @param world     world to write
 * @param filename  The output filename of the HDF5 file
 */
void write_hdf5(const World& world, const std::string& filename, uint64_t iterat

    static H5::H5File file(filename, H5F_ACC_TRUNC);

    H5::Group group(file.createGroup("/" + std::to_string(iteration)));
    write_hdf5_data(group, "world", {world.latitude, world.longitude}, world.dat
}

/** Simulation of a flat word climate
 *
 * @param num_of_iterations  Number of time steps to simulate
 * @param model_filename     The filename of the world model to use (HDF5 file)
 * @param output_filename    The filename of the written world history (HDF5 fil
 */
void simulate(uint64_t num_of_iterations, const std::string& model_filename, con

    // for simplicity, read in full model
    World global_world = read_world_model(model_filename);

    // Finished? TODO: compute offsets according to rank and domain decompositio
    // figure out size of domain for this rank
    const long int offset_longitude = -1 + mpi_rank * global_world.longitude / n
    const long int offset_latitude  = -1;
    const uint64_t longitude = (global_world.longitude) / mpi_size + 2; // No +2
    const uint64_t latitude  = global_world.latitude + 2;
// one ghost cell on each end. Gets ghost cells from own data.


    // copy over albedo data to local world data
    std::vector<double> albedo(longitude*latitude);
    for (uint64_t i = 1; i < latitude-1; ++i)
    for (uint64_t j = 1; j < longitude-1; ++j) {
        uint64_t k_global = (i + offset_latitude) * global_world.longitude
                          + (j + offset_longitude);
        albedo[i * longitude + j] = global_world.albedo_data[k_global];
    }
```

```
// create local world data
World world = World(latitude, longitude, 293.15, albedo);
world.global_latitude  = global_world.latitude;
world.global_longitude = global_world.longitude;
world.offset_latitude  = offset_latitude;
world.offset_longitude = offset_longitude;

// set up counters and loop for num_iterations of integration steps
const double delta_time = world.global_longitude / 36.0;

auto begin = std::chrono::steady_clock::now();
for (uint64_t iteration=0; iteration < num_of_iterations; ++iteration) {
    world.time = iteration / delta_time;
    integrate(world);

    // TODO: gather the Temperature on rank zero
    // remove ghostzones and construct global data from local data
    for (uint64_t i = 1; i < latitude-1; ++i)
    for (uint64_t j = 1; j < longitude-1; ++j) {
        uint64_t k_global = (i + offset_latitude) * global_world.longitude
                          + (j + offset_longitude);
        global_world.data[k_global] = world.data[i * longitude + j];
    }

    // We need only


    if (!output_filename.empty()) {
        // Only rank zero writes water history to file
        if (mpi_rank == 0) {
            write_hdf5(global_world, output_filename, iteration);
            std::cout << iteration << "----";
            stat(global_world);
        }
    }
}
auto end = std::chrono::steady_clock::now();


// if (mpi_rank == 0){ // OBS should all ranks run this?
// }
stat(world);
checksum(world);

if (mpi_rank == 0){
```

```
    std::cout << "elapsed time : " << (end - begin).count() / 1000000000.0 << "
    }
}

/** Main function that parses the command line and start the simulation */
int main(int argc, char **argv) {

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

    // Get the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    std::cout << "Flat World Climate running on " << processor_name
              << ", rank " << mpi_rank << " out of " << mpi_size << std::endl;

    uint64_t iterations=0;
    std::string model_filename;
    std::string output_filename;

    std::vector <std::string> argument({argv, argv+argc});

    for (long unsigned int i = 1; i<argument.size() ; i += 2){
        std::string arg = argument[i];
        if(arg=="-h"){ // Write help
            std::cout << "./fwc --iter <number of iterations>"
                      << " --model <input model>"
                      << " --out <name of output file>\n";
            exit(0);
        } else if (i == argument.size() - 1)
            throw std::invalid_argument("The last argument (" + arg +") must hav
        else if(arg=="--iter"){
            if ((iterations = std::stoi(argument[i+1])) < 0)
                throw std::invalid_argument("iter most be positive (e.g. --iter 1
        } else if(arg=="--model"){
            model_filename = argument[i+1];
        } else if(arg=="--out"){
```

```
            output_filename = argument[i+1];
        } else{
            std::cout << "-->error:the argument type is not recognized\n";
        }
    }
    if (model_filename.empty())
        throw std::invalid_argument("You must specify the model to simulate "
                                    "(e.g. --model models/small.hdf5)");
    if (iterations==0)
        throw std::invalid_argument("You must specify the number of iterations "
                                    "(e.g. --iter 10)");

    simulate(iterations, model_filename, output_filename);

    MPI_Finalize();

    return 0;
}
```