Assignment 5

High Performance Parallel Computing 2022

Kimi Cardoso Kreilgaard & Linea Stausbøll Hedemark

March 2022

1 MPI Parallelise the Program

In parallelising the code we started from the bottom in the `simulate` function. We decided to work by longitude, so we set up the decomposition by defining a local longitude size `loc_x_size`, which is the length of global longitude divided by number of processes. We choose to decompose the longitude as opposed to latitude, as shorter longitude shortens the width of the area that is computed in the processes, and needs less communication between ghost cells. We redefine the local longitudes to include this, with the two added ghost cells. Then we use the `MPI_Gather` call to collect the temperature from local worlds and save it on a vector, `total_data`. When all the data has been gathered, the zeroth rank writes out a file. We then had to redefine the chekcsum calculation, where we use MPI_Reduce to sum all the local world checksums to a global checksum which is printed by rank 0. In `exchange_ghost_cells` we used Send and Receive calls to exchange ghost cells between ranks. We use non-blocking techniques with `MPI_Isend` and `MPI_Irecv` in combination with `MPI_Wait` to make sure the program doesn't stall. The exchanged info is saved in receive_vectors, that we then use to update the local worlds. When using the send and receive calls we use the tag to make sure the exchanges are made correctly. This isn't very important for a number of processes larger than 2, but for a two-split, the ghost cells were not correctly exchanged when we hadn't put in a tag check, so we observed a line down the middle of our world in the visual.ipynb file.
In `stat` we use `MPI_Reduce` to combine all the stats for the local worlds. We make sure to call the right operation, so that we are calculating the minimum, maximum and mean correctly. Everything is collected on global values, that are then printed by rank 0.

2 Strong Scaling Using SLURM

Before we can test the strong scaling of the program, we need to select a large enough number of iterations for each model size. We use 10000 iterations for benchmarking the small model, 1000 iterations for the medium resolution model and 100 iterations for the large resolution model. To see a significant strong scaling, we want to reduce the sequential part of the code, we therefore turn off I/O when benchmarking. We therefore don't gather the temperature data on rank 0 or write to the output file, since this cannot

be parallelised and therefore is not something we want to investigate the lack of scaling of. The strong scaling for respectively the small, medium and large resolution model can be seen in Fig. 1 below. The black line shows the ideal scaling we would observe if there were no sequential part to the code. Even though we have removed the I/O segment of the code, there is still a large sequential part meaning that the strongest scaling we can obtain is around a 12 times speed up for 60 cores. The sequential part remaining consist of send and receive calls, loading in the model and reducing the checksum of each subcomponent into one collective checksum etc. There is a discrepencany in the observed scaling for the small model and the two larger models up to around 30 cores. We believe this is due to the size of the model, that has to be loaded on each rank before the simulation can begin.

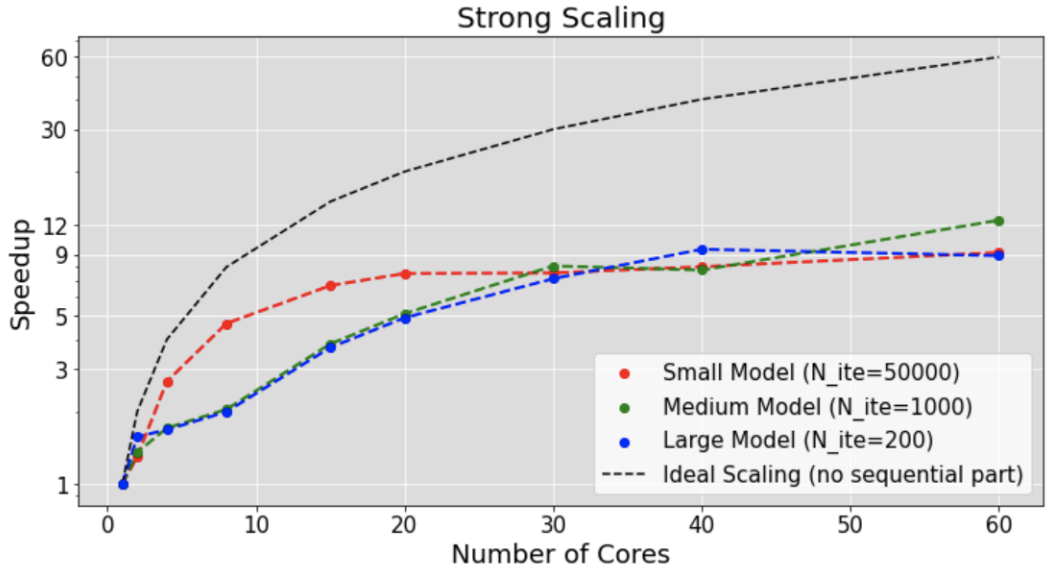

Figure 1: The speedup as a function of number of cores used plotted for respectively the small, medium and large resolution model. The black line shows the ideal scaling one would obtain, if there were no sequential part to the code, and all could be parallelised ideally. Notice that the y-axis is a log scale.

3 The 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;
    // 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 coordinate.
    // See: <https://en.wikipedia.org/wiki/Albedo>
    // NB: this vector has the same length as ‘data‘ and must be interpreted in two dimension as well.
    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 the same temperature).
     * @param albedo_data The measure of the diffuse reflection of solar radiation at each world coordinate.
     * 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(longitude),
                                             data(latitude * longitude, temperature),
                                             albedo_data(std::move(albedo_data)) {}
```

```
};

double checksum(World &world) {
    //
    double cs=0;
    // TODO: make sure checksum is computed globally
    // only loop *inside* data region -- not in ghostzones!

    // we don't change the code here, but compute globally when it is called in simulate

    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];
    }
    return cs;
}

void stat(World &world) {
    // TODO: make sure stats are computed globally
    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];
    }
    //changed global_latitude to latitude, to get local mean
    meant = meant / (world.latitude * world.longitude);

    //sum local values with MPI_Reduce

    //make new values for global, so we don't overwrite locals
    double min_global;
    double max_global;
    double mean_global;

    MPI_Reduce(&mint, &min_global, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
    MPI_Reduce(&maxt, &max_global, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Reduce(&meant, &mean_global, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    //divide by number of processes
    mean_global /= mpi_size;

    if (mpi_rank == 0){
        std::cout << "min: " << min_global
                  << ", max: " << max_global
                  << ", avg: " << mean_global << std::endl;
```

```
    }
}

/** Exchange the ghost cells i.e. copy the second data row and column to the very last data row and column and vice
    ↪ versa.
 *
 * @param world The world to fix the boundaries for.
 */
void exchange_ghost_cells(World &world) {
    // TODO: figure out exchange of ghost cells bewteen ranks
            //completed (ish)

    //Vertical ghost cells

    //define send message vectors
    std::vector<double> emit_right(world.latitude);
    std::vector<double> emit_left(world.latitude);

    for (uint64_t i = 0; i < world.latitude; ++i){
        //update vectors to have the information we would like to send
        emit_right[i + 0] = world.data[i*world.longitude + world.longitude-2];
        emit_left[i + 0] = world.data[i*world.longitude + 1];
    }

    //define postbox vectors to receive messages
    std::vector<double> receive_right(world.latitude);
    std::vector<double> receive_left(world.latitude);

    //Set up request, to enable Isend and Irecv request setup
    MPI_Request req_right, req_left, req_rec_right, req_rec_left;

    //define ranks
    int right_rank = (mpi_rank + 1)%mpi_size;
    int left_rank = (mpi_rank - 1 + mpi_size)%mpi_size;

    //send messages
    MPI_Isend(&emit_right[0], world.latitude, MPI_DOUBLE, right_rank, 1, MPI_COMM_WORLD, &req_right);
    MPI_Isend(&emit_left[0], world.latitude, MPI_DOUBLE, left_rank, 0, MPI_COMM_WORLD, &req_left);

    //receive messages
    MPI_Irecv(&receive_right[0], world.latitude, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD, &req_rec_right);
    MPI_Irecv(&receive_left[0], world.latitude, MPI_DOUBLE, left_rank, 1, MPI_COMM_WORLD, &req_rec_left);

    //wait
    MPI_Wait(&req_right, MPI_STATUS_IGNORE);
    MPI_Wait(&req_left, MPI_STATUS_IGNORE);
    MPI_Wait(&req_rec_right, MPI_STATUS_IGNORE);
    MPI_Wait(&req_rec_left, MPI_STATUS_IGNORE);
```

```
    //store received data
    for (uint64_t i = 0; i < world.latitude; ++i){
        world.data[world.longitude*i + 0] = receive_left[i];
        world.data[(world.longitude-1) + world.longitude*i] = receive_right[i];

        }

    for (uint64_t j = 0; j < world.longitude; ++j) {
        world.data[0*world.longitude + j] = world.data[(world.latitude-2)*world.longitude + j];
        world.data[(world.latitude-1)*world.longitude + j] = world.data[1*world.longitude + j];
        }
}

/** 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.longitude + j]);
        }
    }
    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 + down) / 5.;
            }
        }
        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 of dimension must be two.");
    }
```

```
    if (dataset.getTypeClass() != H5T_FLOAT or dataset.getFloatType().getSize() != 8) {
        throw std::invalid_argument("Error while reading the model: wrong data type, must be double.");
    }

    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, dataspace);
    std::cout << "World model loaded - - latitude: " << (unsigned long) (dims[0]) << ", longitude: "
              << (unsigned long) (dims[1]) << std::endl;
    return World(static_cast<uint64_t>(dims[0]), static_cast<uint64_t>(dims[1]), 293.15, std::move(data_out));
}

/** 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<double>& data) {
    H5::DataSpace dataspace(static_cast<int>(shape.size()), &shape[0]);
    H5::DataSet dataset = group.createDataSet(name.c_str(), H5::PredType::NATIVE_DOUBLE, dataspace);
    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 iteration) {

    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.data);
}

/** 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 file)
 */
void simulate(uint64_t num_of_iterations, const std::string& model_filename, const std::string& output_filename) {

    // for simplicity, read in full model
```

```
World global_world = read_world_model(model_filename);

// TODO: compute offsets according to rank and domain decomposition
// figure out size of domain for this rank

//define size of local longitude splits
const uint64_t loc_x_size = global_world.longitude / mpi_size;
const long int offset_longitude = -1 + mpi_rank*loc_x_size ;
const long int offset_latitude = -1;

//define new local longitude
const uint64_t longitude = loc_x_size + 2;
const uint64_t latitude = global_world.latitude + 2;

// 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

    // Make place to store data in 0
    std::vector<double> total_data;

    if (mpi_rank==0){
        total_data.resize( mpi_size * world.data.size() );
    }

    // This gather all data on rank 0
```

```
    MPI_Gather(&world.data[0], world.data.size(), MPI_DOUBLE, total_data.data(), world.data.size(),
        ↪ MPI_DOUBLE, 0, MPI_COMM_WORLD);


    if (mpi_rank == 0) {

        // Loop over ranks
        for (int r=0; r<mpi_size; ++r){

            // Calc offset latitude for that rank
            long int offset_lon = -1 + r*loc_x_size;

            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_lon);

                global_world.data[k_global] = total_data[world.data.size()*r + i * longitude + j];
                }


            }
            if (!output_filename.empty()) {
                write_hdf5(global_world, output_filename, iteration);
                std::cout << iteration << " ---- ";
                }
        }

    }
    // we changed world to global_world in stat and state to only do it for mpi_rank 0
    //it gave infinite mean tho, bcuz latitudes weren't updated
stat(world);

//start the clock
auto end = std::chrono::steady_clock::now();

//get checksums
double cs = checksum(world);

//define pointer to keep global checksum in
double cs_global;

//collect and sum all local checksums and save in cs_global
MPI_Reduce(&cs, &cs_global, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

//make rank 0 print out the cross checks
if (mpi_rank == 0){
    std::cout << "checksum      : " << cs_global << 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) {

    // 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 have a value");
        else if(arg=="--iter"){
            if ((iterations = std::stoi(argument[i+1])) < 0)
                throw std::invalid_argument("iter most be positive (e.g. -iter 1000)");
        } 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;
}
```