Assignment 2

High Performance Parallel Computing 2022

Kimi Cardoso Kreilgaard & Linea Stausbøll Hedemark

February 2022

1 Task 1: Write a functioning master-worker program

In our program we used MPI_Send and MPI_receive to distribute the tasks. The masters post office ended up in three parts - the first Send element was to send out the first seven tasks to get the workers started, then an intermediate function that received completed tasks and sent out new ones as long as there were still tasks to be completed, and then finally a function that receives the final tasks and tells the workers to shut down. The workers function received their task, completed it and sent it back to the worker, along with their assigned rank as a tag, so that the worker could collect the information on which task was completed and by which worker. We used blocked messaging, but examined how long our program was idle for, i.e. if non-blocking would make a significant improvement on runtime. We found that though there was a little down time of 0.001 s (in the HEP part of the assignment), we found it to be reasonable, and not worth the trouble of implementing non-blocking.

2 Task 2: Master-worker program for HEP data processing

The primary challenge in this task was to remember and find all the places we had put in integer as type instead of double, but once that was dealt with we implemented the same type of program, with the same three-element post office in the master function. The difference being that now the `task_function` used by the worker requires the data as an input and computes an accuracy that is sent back to the master. We validated our results by comparing the best accuracy obtained to the one obtained by the sequential results.

3 Task 3: Strong scaling of HEP processing using SLURM

We wanted to explore different configurations, both configurations from which we expected high performance and configurations we expected to perform poorly. The plan was to test various configurations with 1 or 2 nodes and number of cores ranging from 2 to 128. To demonstrate that communication between different nodes takes more time, we wanted to compare a configuration with 8 nodes and 8 cores to one with 1 node and 8 cores. There was however a problem with ERDA, and we were therefore not successful in using more than 1 node at a time. The configurations we were succesful in running are

Nodes	**Cores**	**Elapsed Time** [s]	**Task Time** [μs]
1	2	168	2563
1	4	64.32	981.5
1	8	30.79	469.8
1	16	15.04	229.6
1	32	7.541	115.1
1	64	3.885	59.28

Table 1: Different SLURM configurations listed with its timing results. All results were obtained with 4 cuts to get a statistically significant result.

noted in Tab. 1 along with the elapsed time and task time produced by the code. To explore another non-typical or non-recommended set up, we attempted a configuration with 1 node and 128 cores. This is because each node has 64 physical cores, but one can enable 128 logical ones (SMT or hyperthreading) which are not physical, allowing more workers to be placed on the same core. Since the cores have to "share" the physical cores, and thus compete for cache, registers and memory bandwidth, we don't expect to see significant changes in the efficiency compared to 1 node, 64 cores. This also becomes evident in the plot in Fig. 1 where there is next to no change in absolute performance, and a more significant change in the task time per worker, since the number of workers doubles, although only half of them can work at a time. Generally what we see in the plot is that the performance improves with more workers, and that the relative performance improves more quickly than the absolute performance. In Fig. 2 the total CPU time is plotted as a function of the cores used. The CPU time is calculated as the product of the elapsed time and the number of workers (the elapsed time is comparable for all of the configurations since all used 4 cuts). In the ideal case this would be a flat curve, although we do expect some fluctuations due to the physical distance between master and worker and the time it takes to send and receive a message. This is also the trend we see in Fig. 2 if we only consider the physical cores. Up to 64 cores the trend is mostly flat, but from 64 to 128 on the x-axis the CPU time almost doubles, which again is because only half of the workers can work at a time. Clearly this is therefore not an efficient set-up, but nonetheless and interesting case.

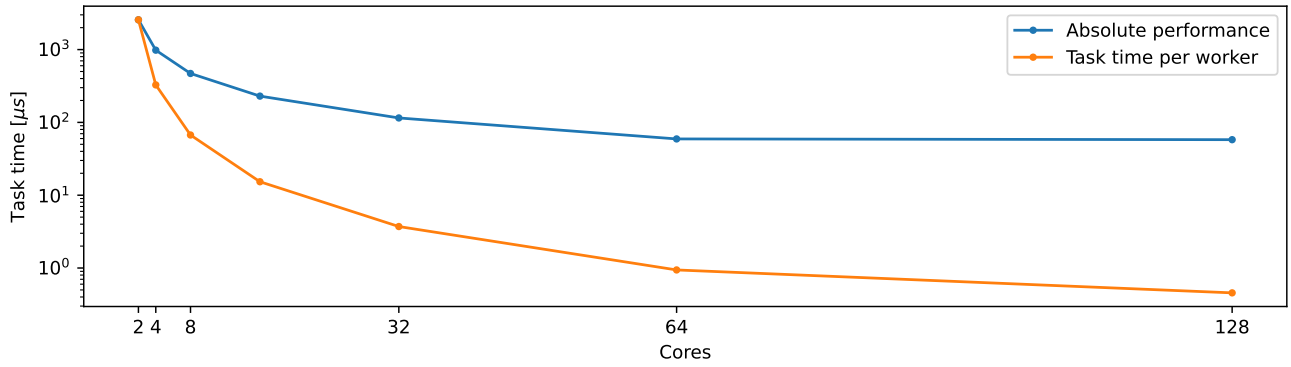

Figure 1: Absolute performance given as task time given by the program, as well as relative performance where absolute perfomance is seen relative to the number of workers (number of cores -1). Performance clearly improves as we utilise all the available cores on the node, but flattens out as we start using more logical cores than are physically available.

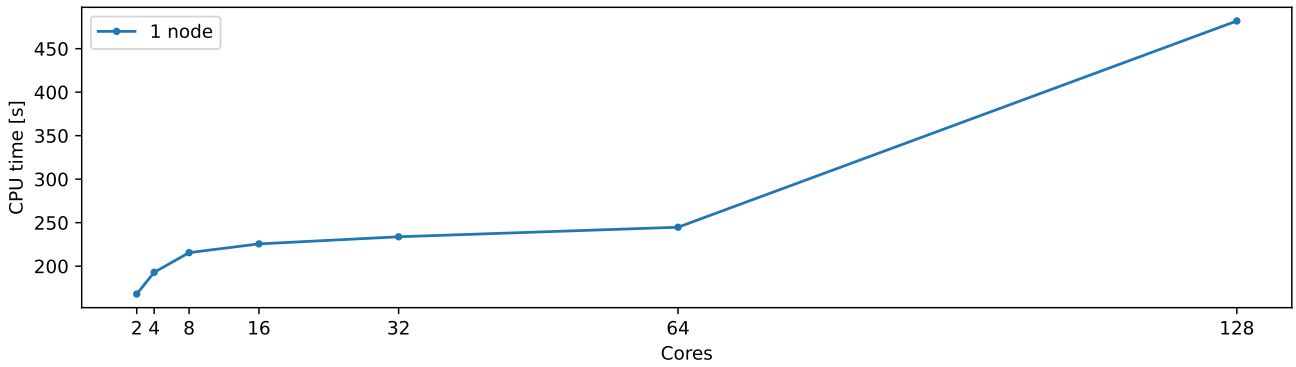

Figure 2: CPU time given as elapsed time multiplied by the number of workers. Curve is relatively flat, with the exception of 127 workers, as there is little improvement between 64 and 128 logical cores, when the physical max of 64 has been reached.

3.1 Notes on Debugging

For a moment we thought that we had a severe problem with our code, since the scaling with the parallization was not at all as expected, with almost no difference and sometimes fewer workers completing the job faster than many times more workers. It turned out that the terminal in ERDA (at least on our computer) does not register that the `job.sh` file has changed even though it has been saved, so we were just running the same configuration again and again. This was solved by closing the terminal between each change to the `job.sh` file. But before we figured this out, we made the following tests:

- We checked that the work was distributed relatively even between the workers, so that performance would increase with number of workers. We did this by adding a counter to the worker loop. When this counter reaches 100, meaning that a worker has completed 100 tasks, this would be printed, indicating that the work in fact is distributed if all workers print this statement.

- We timed how much time a worker "wasted" time waiting for orders. We saved a time stamp right after a worker receives an order, and right after it sends its result back to the master. Subtracting one from the other, we can both see how long the worker takes to complete a task and how long it waits from when it sends the result back to when it receives new instructions. The idea was that if a lot of time was wasted waiting for order because the master was busy, this could reduce the performance, and maybe cause the pattern we were seeing.

4 The Code for Task 1

```
/*
  Assignment: Make an MPI task farm. A "task" is a randomly generated integer.
  To "execute" a task, the worker sleeps for the given number of milliseconds.
  The result of a task should be send back from the worker to the master. It
  contains the rank of the worker
*/

#include <iostream>
#include <random>
#include <chrono>
#include <thread>
#include <array>

// To run an MPI program we always need to include the MPI headers
#include <mpi.h>

/* Define constants and random seed to make it reproducible */
const int NTASKS=5000; // number of tasks
const int RANDOM_SEED=1234;

/* Define the master function, the one who manages the tasks */
void master (int nworker) {

    /* two arrays that we define in one line, static size so not exactly like vectors */
    std::array<int, NTASKS> task, result;

    // set up a random number generator
    std::random_device rd;
    std::default_random_engine engine(RANDOM_SEED);

    // make a distribution of random integers in the interval [0:30]
    std::uniform_int_distribution<int> distribution(0, 30);

    /* Select one random number that represents the task.*/
    for (int& t : task) {
        t = distribution(engine); // set up some "tasks"
    }

    /* Loop over workers to send out first tasks */
    for (int i = 0; i < nworker; i++ ) {

        /* Define task to send */
        int xsend = task[i];

        /* Send out a task to each worker */
        //data_pointer, count, datatype, tag =index, communicator
        MPI_Send(&xsend, 1, MPI_INT, i+1, i, MPI_COMM_WORLD);
```

```
}

/* Define how long we are in the task list */
int task_to_send = nworker;

/* Looping as long as there are tasks to complete */
while (task_to_send < NTASKS) { //IT WAS <= BEFORE

    /* Receive messages */
    //data_pointer, count, datatype, source, tag, comm, status
    int xrecv;
    MPI_Status status;
    MPI_Recv(&xrecv, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    /* Extract tag and source */
    int tag = status.MPI_TAG;
    int source = status.MPI_SOURCE;

    /* Store results */
    result[tag] = xrecv;

    /* Define message to send */
    int xsend = task[task_to_send];

    /* Send message */
    MPI_Send(&xsend, 1, MPI_INT, source, task_to_send, MPI_COMM_WORLD);

    /* +1 to task index thing to complete */
    task_to_send++;
}

/* Receive the last tasks from the workers (since we increment task_to_send before we receive)*/
for (int i = 0; i < nworker; i++ ) {

    /* Receive */
    int xrecv;
    MPI_Status status;
    MPI_Recv(&xrecv, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    /* Extract tag and source */
    int tag = status.MPI_TAG;
    int source = status.MPI_SOURCE;

    /* Store results */
    result[tag] = xrecv;

    /* Send signal to stop. 1 = work is done, 0 = more work to do */
    int xsend = 0;
    MPI_Send(&xsend, 1, MPI_INT, source, NTASKS, MPI_COMM_WORLD); //SIGNAL CHANGED TO NTASKS
```

```
    }

    // Print out a status on how many tasks were completed by each worker
    for (int worker=1; worker<=nworker; worker++) {
        int tasksdone = 0; int workdone = 0;
        for (int itask=0; itask<NTASKS; itask++)
        if (result[itask]==worker) {
            tasksdone++;
            workdone += task[itask];
        }
        std::cout << "Master: Worker " << worker << " solved " << tasksdone <<
                    " tasks\n";
    }
}

// call this function to complete the task. It sleeps for task milliseconds
void task_function(int task) {
    std::this_thread::sleep_for(std::chrono::milliseconds(task));
}

void worker (int rank) {

    /* Define signal to 0, there is work to do when beginning */
    int SIGNAL = 0;

    /* Loop as long as signal is zero */
    while (SIGNAL == 0) {

        /* Receive message from master */
        int xrecv;
        MPI_Status status;
        MPI_Recv(&xrecv, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status); //SET TAG TO NTASK
            ↪ FOR SIGNAL LATER

        /* Extract tag */
        int tag = status.MPI_TAG;

        if (tag == NTASKS){
            return;
        }

        /* Perform task */
        task_function(xrecv);

        /* Send result (rank) to master */
        MPI_Send(&rank, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);

    // CHANGE TO IRECV AND ISEND, WHILE SIGNAL MAYBE WITH TAG FROM MASTER TO WORKER
```

```
    }
}

int main(int argc, char *argv[]) {
    /* allocate memory for them */
    int nrank, rank;

    /* argc stuff is for synchronizing arguments on all ranks */
    MPI_Init(&argc, &argv); // set up MPI
    /* save rank and nrank value at allocated memory */
    MPI_Comm_size(MPI_COMM_WORLD, &nrank); // get the total number of ranks
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get the rank of this process

    /* The same code runs on multiple computers, so first we need to check if the computer it runs on is a master or
        ↪ worker */
    if (rank == 0) // rank 0 is the master
        master(nrank-1); // there is nrank-1 worker processes
    else // ranks in [1:nrank] are workers
        worker(rank);

    MPI_Finalize(); // shutdown MPI
}
```

5 The Code for Task 2 and 3

```
/*
  Assignment: Make an MPI task farm for analysing HEP data
  To "execute" a task, the worker computes the accuracy of a specific set of cuts.
  The resulting accuracy should be send back from the worker to the master.
*/

#include <iostream>
#include <sstream>
#include <fstream>
#include <iomanip>
#include <random>
#include <chrono>
#include <thread>
#include <array>
#include <vector>

// To run an MPI program we always need to include the MPI headers
#include <mpi.h>

// Number of cuts to try out for each event channel.
// BEWARE! Generates n_cuts^8 permutations to analyse.
// If you run many workers, you may want to increase from 3.
const int n_cuts = 4;
const long n_settings = (long) pow(n_cuts,8);
const long NO_MORE_TASKS = n_settings+1;

// Class to hold the main data set together with a bit of statistics
class Data {
public:
    long nevents=0;
    std::string name[8] = { "averageInteractionsPerCrossing", "p_Rhad","p_Rhad1",
                            "p_TRTTrackOccupancy", "p_topoetcone40", "p_eTileGap3Cluster",
                            "p_phiModCalo", "p_etaModCalo" };
    std::vector<std::array<double, 8>> data; // event data
    std::vector<long> NvtxReco; // counters; don't use them
    std::vector<long> p_nTracks;
    std::vector<long> p_truthType; // authorative truth about a signal

    std::vector<bool> signal; // True if p_truthType=2

    std::array<double,8> means_sig {0}, means_bckg {0}; // mean of signal and background for events
    std::array<double,8> flip; // flip sign if background larger than signal for type of event
};

// Routine to read events data from csv file and calculate a bit of statistics
Data read_data() {
    // name of data file
```

```
std::string filename="mc_ggH_16_13TeV_Zee_EGAM1_calocells_16249871.csv";
std::ifstream csvfile(filename); // open file

std::string line;
std::getline(csvfile, line); // skip the first line

Data ds; // variable to hold all data of the file

while (std::getline(csvfile,line)) { // loop over lines until end of file
    if (line.empty()) continue; // skip empty lines
    std::istringstream iss(line);
    std::string element;
    std::array<double,8> data;

    // read in one line of data in to class
    std::getline(iss, element, ','); // line counter, skip it
    std::getline(iss, element, ','); // averageInteractionsPerCrossing
    data[0] = std::stod(element);
    std::getline(iss, element, ','); // NvtxReco
    ds.NvtxReco.push_back(std::stol(element));
    std::getline(iss, element, ','); // p_nTracks
    ds.p_nTracks.push_back(std::stol(element));
    // Load in a loop the 7 next data points:
    // p_Rhad, p_Rhad1, p_TRTTrackOccupancy, p_topoetcone40, p_eTileGap3Cluster, p_phiModCalo, p_etaModCalo
    for(int i=1; i<8; i++) {
        std::getline(iss, element, ',');
        data[i] = std::stod(element);
    }
    std::getline(iss, element, ','); // p_truthType
    ds.p_truthType.push_back(std::stol(element));
    ds.data.push_back(data);
    ds.nevents++;
}

// Calculate means. Signal has p_truthType = 2
ds.signal.resize(ds.nevents);
long nsig=0, nbckg=0;
for (long ev=0; ev<ds.nevents; ev++) {
    ds.signal[ev] = ds.p_truthType[ev] == 2;
    if (ds.signal[ev]) {
        for(int i=0; i<8; i++) ds.means_sig[i] += ds.data[ev][i];
        nsig++;
    } else {
        for(int i=0; i<8; i++) ds.means_bckg[i] += ds.data[ev][i];
        nbckg++;
    }
}
for(int i=0; i<8; i++) {
    ds.means_sig[i] = ds.means_sig[i] / nsig;
```

```
        ds.means_bckg[i] = ds.means_bckg[i] / nbckg;
    }

    // check for flip and change sign of data and means if needed
    for(int i=0; i<8; i++) {
        ds.flip[i]= (ds.means_bckg[i] < ds.means_sig[i]) ? -1 : 1;
        for (long ev=0; ev<ds.nevents; ev++) ds.data[ev][i] *= ds.flip[i];
        ds.means_sig[i] = ds.means_sig[i] * ds.flip[i];
        ds.means_bckg[i] = ds.means_bckg[i] * ds.flip[i];
    }

   return ds;
}

// call this function to complete the task. It calculates the accuracy of a given set of settings
double task_function(std::array<double,8>& setting, Data& ds) {
    // pred evalautes to true if cuts for events are satisfied for all cuts
    std::vector<bool> pred(ds.nevents,true);
    for (long ev=0; ev<ds.nevents; ev++)
        for (int i=0; i<8; i++)
            pred[ev] = pred[ev] and (ds.data[ev][i] < setting[i]);

    // accuracy is percentage of events that are predicted as true signal if and only if a true signal
    double acc=0;
    for (long ev=0; ev<ds.nevents; ev++) acc += pred[ev] == ds.signal[ev];

    return acc / ds.nevents;
}

void master (int nworker, Data& ds) {
    std::array<std::array<double,8>,n_cuts> ranges; // ranges for cuts to explore

    // loop over different event channels and set up cuts
    for(int i=0; i<8; i++) {
        for (int j=0; j<n_cuts; j++)
            ranges[j][i] = ds.means_sig[i] + j * (ds.means_bckg[i] - ds.means_sig[i]) / n_cuts;
    }

    // generate list of all permutations of the cuts for each channel
    std::vector<std::array<double,8>> settings(n_settings);
    for (long k=0; k<n_settings; k++) {
        long div = 1;
        std::array<double,8> set;
        for (int i=0; i<8; i++) {
            long idx = (k / div) % n_cuts;
            set[i] = ranges[idx][i];
            div *= n_cuts;
        }
        settings[k] = set;
```

```
}

// results vector with the accuracy of each set of settings
std::vector<double> accuracy(n_settings);

auto tstart = std::chrono::high_resolution_clock::now(); // start time (nano-seconds)

// ================================================================

/* Loop over workers to send out first tasks */
for (int k = 0; k < nworker; k++ ) {

    /* Send out a task to each worker, i.e. a setting with 8 cuts */
    //data_pointer, count, datatype, source, tag=index, communicator
    // MAYBE THE COUNT SHOULD BE ONE, SINCE ITS IS A VECTOR, WE CHOSE 8 SINCE IT
        ↪ CONTAINS 8 DOUBLES
    MPI_Send(&settings[k], 8, MPI_DOUBLE, k+1, k, MPI_COMM_WORLD);
}

/* Define how far we are in the task list */
int task_to_send = nworker;

/* Looping as long as there are tasks to complete */
while (task_to_send < n_settings) {

    /* Receive messages */
    //data_pointer, count, datatype, source, tag, comm, status
    double xrecv;
    MPI_Status status;
    MPI_Recv(&xrecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    /* Extract tag and source */
    int tag = status.MPI_TAG;
    int source = status.MPI_SOURCE;

    /* Store results */
    accuracy[tag] = xrecv;

    /* Send message */
    MPI_Send(&settings[task_to_send], 8, MPI_DOUBLE, source, task_to_send, MPI_COMM_WORLD);

    /* +1 to task index thing to complete */
    task_to_send++;
}

/* Receive the last tasks from the workers (since we increment task_to_send before we receive)*/
for (int i = 0; i < nworker; i++ ) {

    /* Receive */
```

```
        double xrecv;
        MPI_Status status;
        MPI_Recv(&xrecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

        /* Extract tag and source */
        int tag = status.MPI_TAG;
        int source = status.MPI_SOURCE;

        /* Store results */
        accuracy[tag] = xrecv;

        /* Send signal to stop. 1 = work is done, 0 = more work to do */
        std::array<double, 8> xsend {};
        MPI_Send(&xsend, 8, MPI_DOUBLE, source, n_settings, MPI_COMM_WORLD); //SIGNAL CHANGED TO
            ↪ NTASKS

    }

    // ================================================================

    auto tend = std::chrono::high_resolution_clock::now(); // end time (nano-seconds)
    // diagnostics
    // extract index and value for best accuracy
    double best_accuracy_score=0;
    long idx_best=0;
    for (long k=0; k<n_settings; k++)
        if (accuracy[k] > best_accuracy_score) {
            best_accuracy_score = accuracy[k];
            idx_best = k;
        }

    std::cout << "Best accuracy obtained :" << best_accuracy_score << "\n";
    std::cout << "Final cuts :\n";
    for (int i=0; i<8; i++)
        std::cout << std::setw(30) << ds.name[i] << " : " << settings[idx_best][i]*ds.flip[i] << "\n";

    std::cout << "\n";
    std::cout << "Number of settings:" << std::setw(9) << n_settings << "\n";
    std::cout << "Elapsed time      :" << std::setw(9) << std::setprecision(4)
              << (tend - tstart).count()*1e-9 << "\n";
    std::cout << "task time [mus]   :" << std::setw(9) << std::setprecision(4)
              << (tend - tstart).count()*1e-3 / n_settings << "\n";
}

void worker (int rank, Data& ds) {

    /* Define signal to 0, there is work to do when beginning */
    int SIGNAL = 0;
```

```
    // ADDED COUNTER FOR CHECK
    //int counter = 0;

    // ADD TIMER FOR CHECK
    // auto t_worker = std::chrono::high_resolution_clock::now(); // start time (nano-seconds)

    /* Loop as long as signal is zero */
    while (SIGNAL == 0) {

        /* Receive message from master */
        std::array<double, 8> xrecv;
        MPI_Status status;
        //data_pointer, count, datatype, source, tag, comm, status
        MPI_Recv(&xrecv, 8, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

        // ADD TIMER FOR CHECK
        // auto t_end_1 = std::chrono::high_resolution_clock::now();
        // if (rank == 1 or rank==2) std::cout << "WORKER " << rank << " Received "
        // << (t_end_1 - t_worker).count()*1e-9 << std::endl;

        /* Extract tag */
        int tag = status.MPI_TAG;

        if (tag == n_settings){
            return;
        }

        /* Perform task */
        // MAYBE WE SHOULD NOT DECLARE IT IS A DOUBLE
        double result = task_function(xrecv, ds);

        /* Send result (accuracy) to master */
        MPI_Send(&result, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);

        // ADD TIMER FOR CHECK
        //auto t_end_2 = std::chrono::high_resolution_clock::now();
        // if (rank == 1 or rank==2) std::cout << "WORKER " << rank << " Sent "
        // << (t_end_2 - t_worker).count()*1e-9 << std::endl;

        // ADDED COUNTER FOR CHECK
        //if (counter==100) std::cout<<"I am RANK"<< rank <<"and I reached 100 tasks"<<std::endl;
        //counter++;

    }
}

int main(int argc, char *argv[]) {
    int nrank, rank;
```

```
    MPI_Init(&argc, &argv); // set up MPI
    MPI_Comm_size(MPI_COMM_WORLD, &nrank); // get the total number of ranks
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get the rank of this process

    // All ranks need to read the data
    Data ds = read_data();

    if (rank == 0) // rank 0 is the master
        master(nrank-1, ds); // there is nrank-1 worker processes
    else // ranks in [1:nrank] are workers
        worker(rank, ds);

    MPI_Finalize(); // shutdown MPI
}
```