Assignment 3: Task Farming

Group 16: Anton Marius Nielsen (kjq186), Asmus Tørsleff (qjv778)
& Ida Rye Hellebek (hcx206)

February 2024

1 Write functional task farming program

The basic principle for parallelisation in our program is that master loops through the tasks, sending each to the next available worker. In more detail this is achieved by:

1. Sending out a task to each worker
2. Waiting for a receive from any worker
3. Storing result and sending new task to newly returned worker, until all tasks are sent
4. Receiving last results, before sending a termination signal to returned workers

In this simple case, the data we send from master to worker and vice versa is a reference to a single integer value. The referenced integer value received by master is used to identify the worker, though this could also be done by using the status.

The termination signal is tag = 1, whereas all other messages have tag = 0.

We use blocking messages in all cases here, as there are little computation to do while waiting for a receive request. In a larger setup, with a higher risk of crashing or unavailable workers, it may be beneficial to adapt the setup to non-blocking send requests.

2 Task farming for HEP data

We adapt our implementation of the master-worker program to the HEP data. The adaptions include:

- changing size and type of the data exchanged between master and worker.
- using the status to identify workers
- keeping track of the task index to archive results at corresponding position

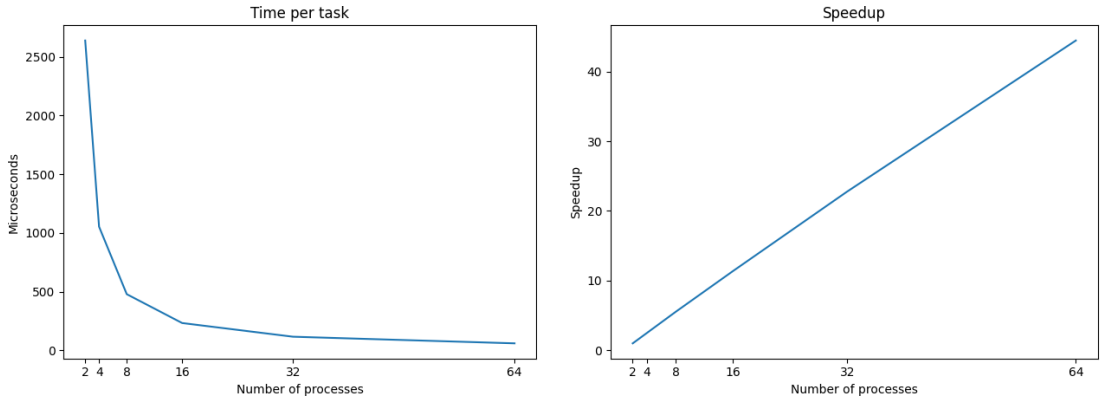
The latter is done by initializing an array of length nworkers for saving the last task index sent to each worker. This is used to archive accuracies in the right order, making sure that we can access the settings used to obtain the best accuracy.

3 Benchmarking on MODI

A: Performance results

With n_cuts = 4, we benchmark the parallel implementation task_farm_HEP.cpp on MODI, with an increasing number of processes. The number of processes shown in figure 1 includes the master, meaning that the number of workers are one less. The data is also shown in appendix table 1.

All cores used are on the same compute node to make the comparison simple. Running on different nodes are expected to yield slightly lower speedup, because the message speed decrease when using the network between nodes.



(a) Performance increase with number of processes (b) Speedup relative to having one worker

Figure 1: Absolute and relative performance

From figure 1b the acieved speedup seems close to ideal speedup, indicating that only a small fraction of the code is serial. However, we know that some part of our code will be serial, and we would not expect ideal speedup for a larger range of processes.

B: Serial and parallel fractions

If we rearrange Amdahl's law, we can use it to calculate the parallel fraction (P) in the program. The baseline here is two processes (one worker, one master), and we use the number of workers as N in the calculations.

$$Speedup(N) = \frac{1}{(1-P) + \frac{P}{N}} \quad \Leftrightarrow \quad P = \frac{Speedup^{-1} - 1}{N^{-1} - 1}$$

Using the speedup when running 64 processes (63 workers), we get a parallel fraction of 99.3% and serial fraction of 0.7%.

A theoretical scaling curve with this percentage is plotted in figure 2. From this, it is obvious that the near ideal speedup only applies with a limited number of processes.

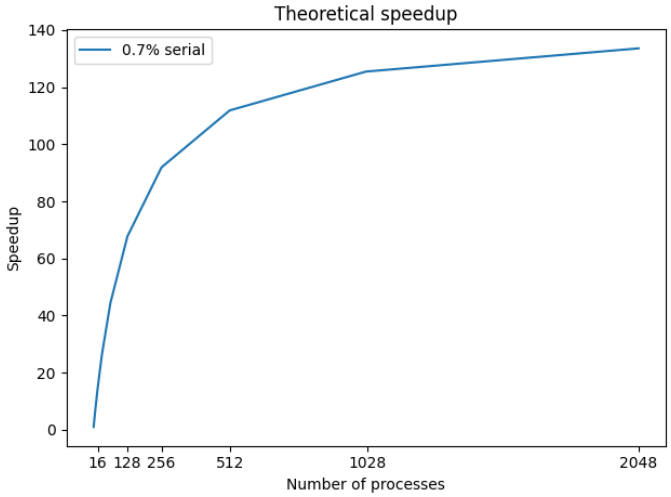


Figure 2: Theoretical scaling curve

C: Discussion of strong scaling

Assuming that the parallel fraction is approximately 99.3%, the gain in performance diminish when using more than a thousand processes. Thus, a maximum of thousand cores may be a reasonable advise for high energy physicists running this analysis.

However, one should be aware of that the hardware architechture for an increasing number of cores might lower the actual speedup. This can be a result of poor network connections and slower memory access. In this case, the most reasonable number of cores might be lower, depending on the exact hardware used.

4 Appendix

Table 1: Performance as total elapsed time and time per task

Number of processes	Seconds elapsed	µ seconds / task
2	172.9	2638
4	69.02	1053
8	31.33	478
16	15.22	232.3
32	7.59	115.8
64	3.89	59.3

5 Implementation 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>
#include <stack>

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

const int NTASKS= 100; //5000;  // number of tasks
const int RANDOM_SEED=1234;

void master (int nworker) {
    std::array<int, NTASKS> task, result;
    std::vector<int> workers(nworker+1, 0); // vector of curr. k
       assigned to worker rank = index


    // set up a random number generator
    std::random_device rd;
    //std::default_random_engine engine(rd());
    std::default_random_engine engine;
    engine.seed(RANDOM_SEED);
    // make a distribution of random integers in the interval [0:30]
    std::uniform_int_distribution<int> distribution(0, 1000);

    // initialize list of tasks
    for (int& t : task)   t = distribution(engine);

    // initialize list of available workers
    std::stack<int> avail_ws;
    for (int w=1; w <= nworker; w++)  avail_ws.push(w);

    // loop through tasks and send out to available workers
    int w;
```

```
    int data;
    MPI_Status status;

    for (int tidx=0; tidx < NTASKS; tidx++) {

        if (!avail_ws.empty()) {
            w = avail_ws.top();
            avail_ws.pop();
        }
        else {
            MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD, &status);
            w = data;
            result[workers[w]] = data;
        }
        MPI_Send(&task[tidx], 1, MPI_INT, w, 0, MPI_COMM_WORLD);
        workers[w] = tidx;
    }

    // send termination signal to all workers
    for (int w_count=0; w_count < nworker; w_count++) {
        MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &
            status);
        w = data;
        result[workers[w]] = data;
        MPI_Send(&task[0], 1, MPI_INT, w, 1, MPI_COMM_WORLD);
    }

    // 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) {

    int data = 0;
    MPI_Status status;
    while (true) {
        MPI_Recv(&data, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
            status);
        if (status.MPI_TAG == 1) break;
        task_function(data);
        MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
}

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

    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
}
```

6 Implementation for task 2

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

// 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 = 3;
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::cout << "-nworker-" << nworker << std::endl;
    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)

// ====================/ Our implementation of master-worker
    /====================

// initialize list of available workers
std::stack<int> avail_ws;
for (int w=1; w <= nworker; w++)  avail_ws.push(w);

// start sending out tasks
int w; // current worker rank
double acc; // temp loc for computed acc
MPI_Status status;
std::vector<int> workers(nworker+1, 0); // vector of curr. k
    assigned to worker rank = index

for (long k = 0; k < n_settings; k++) {

    if (!avail_ws.empty()) {
        w = avail_ws.top();
        avail_ws.pop();
    }
    else {
        MPI_Recv(&acc, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0,
            MPI_COMM_WORLD, &status);
```

```
            w = status.MPI_SOURCE;
            accuracy[workers[w]] = acc;
        }
        MPI_Send(settings[k].data(), 8, MPI_DOUBLE, w, 0, MPI_COMM_WORLD
            );
        workers[w] = k;
    }

    // send termination signal, here tag=1
    for (int w_count=0; w_count < nworker; w_count++) {
        MPI_Recv(&acc, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
            &status);
        w = status.MPI_SOURCE;
        accuracy[workers[w]] = acc;
        MPI_Send(settings[0].data(), 8, MPI_DOUBLE, w, 1, MPI_COMM_WORLD
            );
    }

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

    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) {

    std::array<double,8> set;
    double acc;
    MPI_Status status;
    while (true) {
        MPI_Recv(&set, 8, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
            status);
        if (status.MPI_TAG == 1) break;
        else {
            acc = task_function(set, ds);
            MPI_Send(&acc, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }
    }
}

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
}
```