Bubble Sort and Linear Regression with MPI

Claudio Scheer[1] and Gabriell Araujo[1]

[1]Master's Degree in Computer Science - PUCRS
{*claudio.scheer, grabriell.araujo*}*@edu.pucrs.br*

October 14, 2020

asdfas w Include a title and one or two paragraphs describing what you plan to do. Tell what interviews, site visits, or other activity you plan. Be specific if you can. Include one good reference you plan to used. This is an example of how to include a citation.

0.1 Introduction

Include a title and one or two paragraphs describing what you plan to do. Tell what interviews, site visits, or other activity you plan. Be specific if you can. Include one good reference you plan to used. This is an example of how to include a citation.

0.2 Background

Give a background on your topic. Include references.

0.3 Results

Results of your interviews or observations. Use information and/or quotes from your interview or observations.

0.4 Discussion

Your comments or evaluation of interview or observations

0.5 Summary

Summarize your paper.

Appendices

Appendix A

Bubble Sort Source Code

```cpp
#include <iostream>
#include <vector>

using namespace std;

namespace dataset {
vector<int> get_vector(int vector_size) {
    vector<int> v;
    for (int i = 0; i < vector_size; i++) {
        v.push_back(vector_size - i);
    }
    return v;
}

vector<vector<int>> get_dataset(int number_vectors, int vector_size) {
    vector<vector<int>> vectors;
    vector<int> v = get_vector(vector_size);
    for (int i = 0; i < number_vectors; i++) {
        vectors.push_back(v);
    }
    return vectors;
}
} // namespace dataset
```

Listing A.1: Dataset generator

```cpp
#include "dataset-generator.cpp"
#include <chrono>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <sstream>
#include <tuple>
#include <vector>

using namespace std;

vector<vector<int>> load_dataset(int number_vectors, int vector_size) {
    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    vector<vector<int>> vectors =
        dataset::get_dataset(number_vectors, vector_size);
    chrono::steady_clock::time_point end = chrono::steady_clock::now();
    double total_time =
        chrono::duration_cast<chrono::duration<double>>(end - begin).count();
    cout << "Time load dataset (s): " << total_time << endl;
    return vectors;
}

vector<int> bubble_sort(vector<int> v) {
    int n = v.size();
    int c = 0;
    int temp;
    int swapped = 1;

    while ((c < (n - 1)) & swapped) {
        swapped = 0;
        for (int d = 0; d < n - c - 1; d++)
            if (v.at(d) > v.at(d + 1)) {
                temp = v.at(d);
                v.at(d) = v.at(d + 1);
                v.at(d + 1) = temp;
                swapped = 1;
            }
        c++;
    }

    return v;
}

int main(int argc, char **argv) {
    int number_vectors = atoi(argv[1]);
    int vector_size = atoi(argv[2]);
    vector<vector<int>> vectors = load_dataset(number_vectors, vector_size);

    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    for (int i = 0; i < vectors.size(); i++) {
        vector<int> v = vectors.at(i);
        vector<int> v_sorted = bubble_sort(v);
    }
    chrono::steady_clock::time_point end = chrono::steady_clock::now();
    double total_time =
        chrono::duration_cast<chrono::duration<double>>(end - begin).count();

    cout << "Number vectors: " << number_vectors << endl;
    cout << "Vector size: " << vector_size << endl;
    cout << "Time sort (s): " << total_time << endl;
    return 0;
}
```

Listing A.2: Bubble Sort Sequential

```cpp
#include "dataset-generator.cpp"
#include <chrono>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <mpi.h>
#include <sstream>
#include <tuple>
#include <vector>

using namespace std;

vector<vector<int>> load_dataset(int number_vectors, int vector_size) {
    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    vector<vector<int>> vectors =
        dataset::get_dataset(number_vectors, vector_size);
    chrono::steady_clock::time_point end = chrono::steady_clock::now();
    double total_time =
        chrono::duration_cast<chrono::duration<double>>(end - begin).count();
    cout << "Time load dataset (s): " << total_time << endl;
    return vectors;
}

vector<int> bubble_sort(vector<int> v) {
    int n = v.size();
    int c = 0;
    int temp;
    int swapped = 1;

    while ((c < (n - 1)) & swapped) {
        swapped = 0;
        for (int d = 0; d < n - c - 1; d++)
            if (v.at(d) > v.at(d + 1)) {
                temp = v.at(d);
                v.at(d) = v.at(d + 1);
```

```
                v.at(d + 1) = temp;
                swapped = 1;
            }
        c++;
    }

    return v;
}

int main(int argc, char **argv) {
    int number_vectors = atoi(argv[1]);
    int vector_size = atoi(argv[2]);

    int vector_tag = 1;
    int kill_tag = 2;
    int request_vector_tag = 3;

    MPI_Status status;
    int my_rank;
    int num_processes;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);

    if (my_rank != 0) {
        int master = 0;
        int ask_for_message = 1;
        int kill_flag = 0;
        while (!kill_flag) {
            if (ask_for_message) {
                // Will only send a new request when the last request was
                // already processed.
                MPI_Send(&ask_for_message, 1, MPI_INT, master,
                        request_vector_tag, MPI_COMM_WORLD);
                ask_for_message = 0;
            }
            // Test whether the master submitted a new job.
            int has_message = 0;
            MPI_Iprobe(master, vector_tag, MPI_COMM_WORLD, &has_message,
                    &status);
            if (has_message) {
                vector<int> v;
                v.resize(vector_size);
                MPI_Recv(&v[0], vector_size, MPI_INT, master, vector_tag,
                        MPI_COMM_WORLD, &status);

                vector<int> v_sorted = bubble_sort(v);
                MPI_Send(&v_sorted, vector_size, MPI_INT, master, vector_tag,
                        MPI_COMM_WORLD);

                ask_for_message = 1;
            }
            // Check for a 'suicide' request.
            MPI_Iprobe(master, kill_tag, MPI_COMM_WORLD, &kill_flag, &status);
        }
    } else {
        vector<vector<int>> vectors = load_dataset(number_vectors, vector_size);

        double begin = MPI_Wtime();

        // Store async requests received from workers.
        vector<MPI_Request> receive_requests(number_vectors);
        vector<vector<int>> ordered_vectors(number_vectors);

        int worker_request = 0;
        for (int i = 0; i < vectors.size(); i++) {
            vector<int> v = vectors.at(i);
            MPI_Recv(&worker_request, 1, MPI_INT, MPI_ANY_SOURCE,
                    request_vector_tag, MPI_COMM_WORLD, &status);
            // Send the vector to the worker.
            MPI_Send(&v, vector_size, MPI_INT, status.MPI_SOURCE, vector_tag,
                    MPI_COMM_WORLD);

            ordered_vectors[i].resize(vector_size);
            MPI_Irecv(&ordered_vectors[i][0], vector_size, MPI_INT,
                     status.MPI_SOURCE, vector_tag, MPI_COMM_WORLD,
                     &receive_requests[i]);
        }

        // Wait for all requests.
        for (int i = 0; i < vectors.size(); i++) {
            MPI_Wait(&receive_requests.at(i), &status);
        }

        // Kill all workers.
        int kill_value = 1;
        for (int i = 1; i < num_processes; i++) {
            MPI_Send(&kill_value, 1, MPI_INT, i, kill_tag, MPI_COMM_WORLD);
        }

        double end = MPI_Wtime();
        double total_time = end - begin;

        cout << "Number processes: " << num_processes << endl;
        cout << "Number vectors: " << number_vectors << endl;
        cout << "Vector size: " << vector_size << endl;
        cout << "Time sort (s): " << total_time << endl;
    }
    MPI_Finalize();

    return 0;
}
```

Listing A.3: Bubble Sort MPI

Appendix B

Linear Regression Source Code

```cpp
#include <iostream>
#include <vector>

using namespace std;

namespace dataset {
vector<int> get_vector(int vector_size) {
    vector<int> v;
    for (int i = 0; i < vector_size; i++) {
        v.push_back(vector_size - i);
    }
    return v;
}

vector<vector<int>> get_dataset(int number_vectors, int vector_size) {
    vector<vector<int>> vectors;
    vector<int> v = get_vector(vector_size);
    for (int i = 0; i < number_vectors; i++) {
        vectors.push_back(v);
    }
    return vectors;
}
} // namespace dataset
```

Listing B.1: Dataset generator

```cpp
#include "dataset-generator.cpp"
#include <chrono>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <sstream>
#include <tuple>
#include <vector>

using namespace std;

vector<dataset::Point> load_dataset(unsigned long long int number_points) {
    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    vector<dataset::Point> points = dataset::get_dataset(number_points);
    chrono::steady_clock::time_point end = chrono::steady_clock::now();
    double total_time = chrono::duration<double>(end - begin).count();
    cout << "Time load dataset (s): " << total_time << endl;
    return points;
}

tuple<double, double, double> execute_lr(vector<dataset::Point> points) {
    chrono::steady_clock::time_point begin = chrono::steady_clock::now();

    unsigned long long int x_sum = 0;
    unsigned long long int y_sum = 0;
    unsigned long long int x_squared_sum = 0;
    unsigned long long int xy_sum = 0;
    int n = (int)points.size();

    for (unsigned long long int i = 0; i < n; i++) {
        int x_aux = points.at(i).x;
        int y_aux = points.at(i).y;

        x_sum += x_aux;
        y_sum += y_aux;
        x_squared_sum += x_aux * x_aux;
        xy_sum += x_aux * y_aux;
    }

    chrono::steady_clock::time_point end = chrono::steady_clock::now();
    double total_time = chrono::duration<double>(end - begin).count();

    double slope = ((double)(n * xy_sum - x_sum * y_sum)) /
                ((double)(n * x_squared_sum - x_sum * x_sum));
    double intercept = ((double)(y_sum - slope * x_sum)) / n;

    return make_tuple(total_time, slope, intercept);
}

int main(int argc, char **argv) {
    unsigned long long int number_points = atoll(argv[1]);
    vector<dataset::Point> points = load_dataset(number_points);
    tuple<double, double, double> results = execute_lr(points);

    double total_time = get<0>(results);
    double slope = get<1>(results);
    double intercept = get<2>(results);
    cout << "Time linear regression (s): " << total_time << endl;
    cout << "Slope: " << slope << endl;
    cout << "Intercept: " << intercept << endl;

    return 0;
}
```

Listing B.2: Linear Regression Sequential

```cpp
#include "dataset-generator.cpp"
#include <chrono>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <mpi.h>
#include <sstream>
#include <tuple>
#include <vector>

using namespace std;

// Store the results from each worker.
struct RegressionSubResults {
    unsigned long long int x_sum;
    unsigned long long int y_sum;
    unsigned long long int x_squared_sum;
    unsigned long long int xy_sum;
};

vector<dataset::Point> load_dataset(unsigned long long int number_points) {
    double begin = MPI_Wtime();
    vector<dataset::Point> points = dataset::get_dataset(number_points);
    double end = MPI_Wtime();
    double total_time = end - begin;
    cout << "Time load dataset (s): " << total_time << endl;
    return points;
}

// Perform linear regression on the subvector.
RegressionSubResults execute_lr(vector<dataset::Point> points) {
```

```
    unsigned long long int x_sum = 0;
    unsigned long long int y_sum = 0;
    unsigned long long int x_squared_sum = 0;
    unsigned long long int xy_sum = 0;
    int n = (int)points.size();

    for (unsigned long long int i = 0; i < n; i++) {
        int x_aux = points.at(i).x;
        int y_aux = points.at(i).y;

        x_sum += x_aux;
        y_sum += y_aux;

        x_squared_sum += x_aux * x_aux;
        xy_sum += x_aux * y_aux;
    }

    return {
        .x_sum = x_sum,
        .y_sum = y_sum,
        .x_squared_sum = x_squared_sum,
        .xy_sum = xy_sum,
    };
}

int main(int argc, char **argv) {
    unsigned long long int number_points = atoll(argv[1]);
    unsigned long long int granularity = atoll(argv[2]);

    int vector_tag = 1;
    int kill_tag = 2;
    int request_vector_tag = 3;

    int number_grains = number_points / granularity;
    MPI_Status status;
    int my_rank;
    int num_processes;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);

    if ((number_points % granularity) > 0) {
        // This avoids the need to deal with the last elements of the array.
        cout << "Error: granularity must be a multiple of the number of points."
             << endl;
        MPI::COMM_WORLD.Abort(-1);
    }

    // Commit Point struct to MPI.
    MPI_Datatype MPI_POINT_TYPE;
    int block_lengths_point[2] = {1, 1};
    MPI_Aint displacements_point[2] = {offsetof(dataset::Point, x),
                                       offsetof(dataset::Point, y)};
    MPI_Datatype types_point[2] = {MPI_INT, MPI_INT};
    MPI_Type_create_struct(2, block_lengths_point, displacements_point,
                           types_point, &MPI_POINT_TYPE);
    MPI_Type_commit(&MPI_POINT_TYPE);

    // Commit RegressionSubResults struct to MPI.
    MPI_Datatype MPI_REGRESSION_SUB_RESULTS_TYPE;
    int block_lengths_regression_sub_results[4] = {1, 1, 1, 1};
    MPI_Aint displacements_regression_sub_results[4] = {
        offsetof(RegressionSubResults, x_sum),
        offsetof(RegressionSubResults, y_sum),
        offsetof(RegressionSubResults, x_squared_sum),
        offsetof(RegressionSubResults, xy_sum)};
    MPI_Datatype types_regression_sub_results[4] = {
        MPI_LONG_LONG_INT, MPI_LONG_LONG_INT, MPI_LONG_LONG_INT,
        MPI_LONG_LONG_INT};
    MPI_Type_create_struct(4, block_lengths_regression_sub_results,
                           displacements_regression_sub_results,
                           types_regression_sub_results,
                           &MPI_REGRESSION_SUB_RESULTS_TYPE);
    MPI_Type_commit(&MPI_REGRESSION_SUB_RESULTS_TYPE);

    if (my_rank != 0) {
        int master = 0;
        int ask_for_message = 1;
        int kill_flag = 0;
        while (!kill_flag) {
            if (ask_for_message) {
                // Will only send a new request when the last request was
                // already processed.
                MPI_Send(&ask_for_message, 1, MPI_INT, master,
                         request_vector_tag, MPI_COMM_WORLD);
                ask_for_message = 0;
            }
            // Test whether the master submitted a new job.
            int has_message = 0;
            MPI_Iprobe(master, vector_tag, MPI_COMM_WORLD, &has_message,
                       &status);
            if (has_message) {
                vector<dataset::Point> points;
                points.resize(granularity);
                MPI_Recv(&points[0], granularity, MPI_POINT_TYPE, master,
                         vector_tag, MPI_COMM_WORLD, &status);

                RegressionSubResults sub_results = execute_lr(points);
                MPI_Send(&sub_results, 1, MPI_REGRESSION_SUB_RESULTS_TYPE,
                         master, vector_tag, MPI_COMM_WORLD);

                ask_for_message = 1;
            }
            // Check for a 'suicide' request.
            MPI_Iprobe(master, kill_tag, MPI_COMM_WORLD, &kill_flag, &status);
        }
    } else {
        vector<dataset::Point> points = load_dataset(number_points);

        double begin = MPI_Wtime();

        // Store async requests received from workers.
        vector<MPI_Request> receive_requests(number_grains);
        vector<RegressionSubResults> regression_sub_results(number_grains);

        int grain = 0;
        int worker_request = 0;
        while (number_points > (grain * granularity)) {
            MPI_Recv(&worker_request, 1, MPI_INT, MPI_ANY_SOURCE,
                     request_vector_tag, MPI_COMM_WORLD, &status);
            // Send the next elements from the dataset to the worker.
            MPI_Send(&points[(grain * granularity)], granularity,
                     MPI_POINT_TYPE, status.MPI_SOURCE, vector_tag,
                     MPI_COMM_WORLD);
            MPI_Irecv(&regression_sub_results[grain], 1,
                      MPI_REGRESSION_SUB_RESULTS_TYPE, status.MPI_SOURCE,
                      vector_tag, MPI_COMM_WORLD, &receive_requests[grain]);
            grain++;
        }

        RegressionSubResults results = {
            .x_sum = 0,
            .y_sum = 0,
            .x_squared_sum = 0,
            .xy_sum = 0,
        };
        // Collect the results of all workers.
        for (int i = 0; i < number_grains; i++) {
            MPI_Wait(&receive_requests.at(i), &status);
            RegressionSubResults sub_results = regression_sub_results.at(i);
            results.x_sum += sub_results.x_sum;
            results.y_sum += sub_results.y_sum;
            results.x_squared_sum += sub_results.x_squared_sum;
            results.xy_sum += sub_results.xy_sum;
        }

        // Kill all workers.
        int kill_value = 1;
        for (int i = 1; i < num_processes; i++) {
            MPI_Send(&kill_value, 1, MPI_INT, i, kill_tag, MPI_COMM_WORLD);
        }

        double end = MPI_Wtime();
        double total_time = end - begin;

        double slope = ((double)(number_points * results.xy_sum -
                                 results.x_sum * results.y_sum)) /
                       ((double)(number_points * results.x_squared_sum -
                                 results.x_sum * results.x_sum));
        double intercept =
            ((double)(results.y_sum - slope * results.x_sum)) / number_points;
        cout << "Time linear regression (s): " << total_time << endl;
        cout << "Slope: " << slope << endl;
        cout << "Intercept: " << intercept << endl;
    }
    MPI_Finalize();

    return 0;
}
```

Listing B.3: Linear Regression MPI