

Bubble Sort and Linear Regression with MPI

Claudio Scheer¹ and Gabriell Araujo¹

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

October 15, 2020

0.1 General Setup

Instead of using the LAD access provided by the professor, we ran our *batch job* on two nodes in the Cer rado cluster. That is because we developed in C++17 and needed a newer version of GCC and OpenMPI than the one provided by LAD, and we already had a *batch job* configured from previous works.

All experiments were executed three times and then the average execution time and the standard deviation were calculated. For the implementation using MPI, we used the master-slave architecture. In short, the slave asks the master for a job, the master sends the job to the slave, the slave processes the job and returns the result. The master waits for the slave's results using an asynchronous call. Finally, when all jobs are completed, the master waits for all the asynchronous results of the slaves and asks the slave to 'commit suicide'¹.

0.2 Bubble Sort

The bubble sort problem addressed here consists of sorting 1000 vectors with 2500 integers. Each slave receives a vector to sort and return the sorted vector to the master. Figure 1 shows the

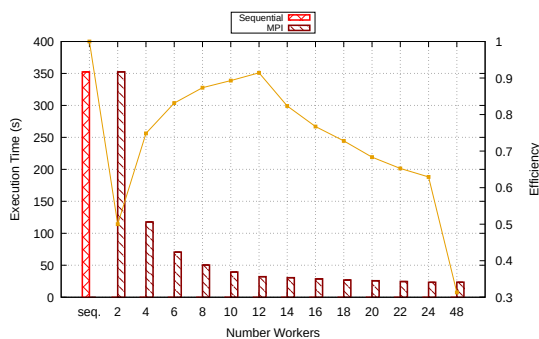


Figure 1: Execution Time x Efficiency

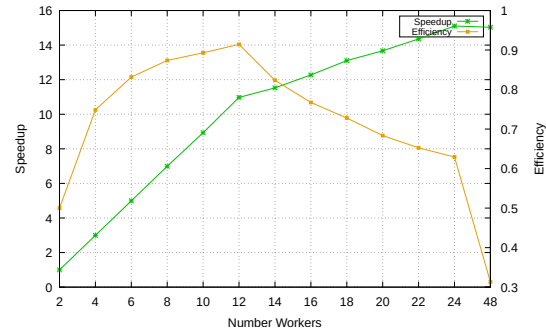


Figure 2: Speedup x Efficiency

0.3 Linear Regression

Linear regression is an algorithm used for predictive analysis. In summary, the algorithm finds a relationship between x and y and can predict a new y using as input a x not yet known by the model. To test the algorithm, we used 100000000 x and y points.

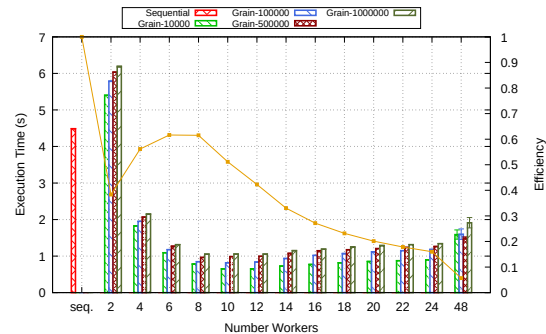


Figure 3: Execution Time x Efficiency

0.4 Results

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

¹What a horrible scenario!

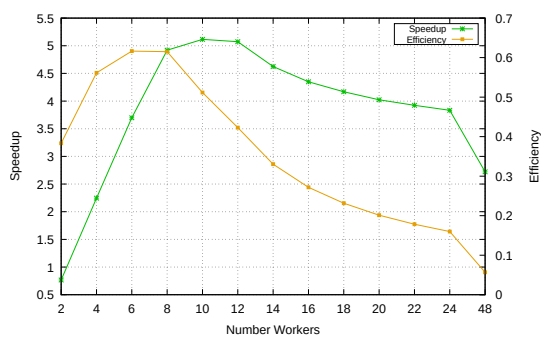


Figure 4: Speedup x Efficiency

Appendices

Appendix A

Bubble Sort Source Code

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 namespace dataset {
7     vector<int> get_vector(int vector_size) {
8         vector<int> v;
9         for (int i = 0; i < vector_size; i++) {
10             v.push_back(vector_size - i);
11         }
12         return v;
13     }
14
15     vector<vector<int>> get_dataset(int number_vectors, int vector_size) {
16         vector<vector<int>> vectors;
17         vector<int> v = get_vector(vector_size);
18         for (int i = 0; i < number_vectors; i++) {
19             vectors.push_back(v);
20         }
21         return vectors;
22     }
23 } // namespace dataset
```

Listing A.1: Dataset generator

```
1 #include "dataset-generator.cpp"
2 #include <chrono>
3 #include <stdio>
4 #include <fstream>
5 #include <iostream>
6 #include <sstream>
7 #include <tuple>
8 #include <vector>
9
10 using namespace std;
11
12 vector<vector<int>> load_dataset(int number_vectors, int vector_size) {
13     chrono::steady_clock::time_point begin = chrono::steady_clock::now();
14     vector<vector<int>> vectors =
15         dataset::get_dataset(number_vectors, vector_size);
16     chrono::steady_clock::time_point end = chrono::steady_clock::now();
17     double total_time =
18         chrono::duration_cast<chrono::duration<double>>(end - begin).count();
19     cout << "Time load dataset (s): " << total_time << endl;
20     return vectors;
21 }
22
23 vector<int> bubble_sort(vector<int> v) {
24     int n = v.size();
25     int c = 0;
26     int temp;
27     int swapped = 1;
28
29     while ((c < (n - 1)) & swapped) {
30         swapped = 0;
31         for (int d = 0; d < n - c - 1; d++)
32             if (v.at(d) > v.at(d + 1)) {
33                 temp = v.at(d);
34                 v.at(d) = v.at(d + 1);
35                 v.at(d + 1) = temp;
36                 swapped = 1;
37             }
38     }
```

```
38     c++;
39 }
40
41     return v;
42 }
43
44 int main(int argc, char **argv) {
45     int number_vectors = atoi(argv[1]);
46     int vector_size = atoi(argv[2]);
47     vector<vector<int>> vectors = load_dataset(number_vectors, vector_size);
48
49     chrono::steady_clock::time_point begin = chrono::steady_clock::now();
50     for (int i = 0; i < vectors.size(); i++) {
51         vector<int> v = vectors.at(i);
52         vector<int> v_sorted = bubble_sort(v);
53     }
54     chrono::steady_clock::time_point end = chrono::steady_clock::now();
55     double total_time =
56         chrono::duration_cast<chrono::duration<double>>(end - begin).count();
57
58     cout << "Number vectors: " << number_vectors << endl;
59     cout << "Vector size: " << vector_size << endl;
60     cout << "Time sort (s): " << total_time << endl;
61     return 0;
62 }
```

Listing A.2: Bubble Sort Sequential

```
1 #include "dataset-generator.cpp"
2 #include <chrono>
3 #include <stdio>
4 #include <fstream>
5 #include <iostream>
6 #include <mpi.h>
7 #include <sstream>
8 #include <tuple>
9 #include <vector>
10
11 using namespace std;
12
13 vector<vector<int>> load_dataset(int number_vectors, int vector_size) {
14     chrono::steady_clock::time_point begin = chrono::steady_clock::now();
15     vector<vector<int>> vectors =
16         dataset::get_dataset(number_vectors, vector_size);
17     chrono::steady_clock::time_point end = chrono::steady_clock::now();
18     double total_time =
19         chrono::duration_cast<chrono::duration<double>>(end - begin).count();
20     cout << "Time load dataset (s): " << total_time << endl;
21     return vectors;
22 }
23
24 vector<int> bubble_sort(vector<int> v) {
25     int n = v.size();
26     int c = 0;
27     int temp;
28     int swapped = 1;
29
30     while ((c < (n - 1)) & swapped) {
31         swapped = 0;
32         for (int d = 0; d < n - c - 1; d++)
33             if (v.at(d) > v.at(d + 1)) {
34                 temp = v.at(d);
35                 v.at(d) = v.at(d + 1);
```

```

36         v.at(d + 1) = temp;
37         swapped = 1;
38     }
39     c++;
40 }
41
42 return v;
43 }
44
45 int main(int argc, char **argv) {
46     int number_vectors = atoi(argv[1]);
47     int vector_size = atoi(argv[2]);
48
49     int vector_tag = 1;
50     int kill_tag = 2;
51     int request_vector_tag = 3;
52
53     MPI_Status status;
54     int my_rank;
55     int num_processes;
56
57     MPI_Init(&argc, &argv);
58     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
59     MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
60
61     if (my_rank != 0) {
62         int master = 0;
63         int ask_for_message = 1;
64         int kill_flag = 0;
65         while (!kill_flag) {
66             if (ask_for_message) {
67                 // Will only send a new request when the last request was
68                 // already processed.
69                 MPI_Send(&ask_for_message, 1, MPI_INT, master,
70                     request_vector_tag, MPI_COMM_WORLD);
71                 ask_for_message = 0;
72             }
73             // Test whether the master submitted a new job.
74             int has_message = 0;
75             MPI_Iprobe(master, vector_tag, MPI_COMM_WORLD, &has_message,
76                 &status);
77             if (has_message) {
78                 vector<int> v;
79                 v.resize(vector_size);
80                 MPI_Recv(&v[0], vector_size, MPI_INT, master, vector_tag,
81                     MPI_COMM_WORLD, &status);
82
83                 vector<int> v_sorted = bubble_sort(v);
84                 MPI_Send(&v_sorted[0], vector_size, MPI_INT, master, vector_tag,
85                     MPI_COMM_WORLD);
86
87                 ask_for_message = 1;
88             }
89             // Check for a 'suicide' request.
90             MPI_Iprobe(master, kill_tag, MPI_COMM_WORLD, &kill_flag, &status);
91         }
92     } else {
93         vector<vector<int>> vectors = load_dataset(number_vectors, vector_size);
94
95         double begin = MPI_Wtime();
96
97         // Store async requests received from workers.
98         vector<MPI_Request> receive_requests(number_vectors);
99         vector<vector<int>> ordered_vectors(number_vectors);
100
101         int worker_request = 0;
102         for (int i = 0; i < vectors.size(); i++) {
103             vector<int> v = vectors.at(i);
104             MPI_Recv(&worker_request, 1, MPI_INT, MPI_ANY_SOURCE,
105                 request_vector_tag, MPI_COMM_WORLD, &status);
106             // Send the vector to the worker.
107             MPI_Send(&v[0], vector_size, MPI_INT, status.MPI_SOURCE, vector_tag,
108                 MPI_COMM_WORLD);
109
110             ordered_vectors[i].resize(vector_size);
111             MPI_Irecv(&ordered_vectors[i][0], vector_size, MPI_INT,
112                 status.MPI_SOURCE, vector_tag, MPI_COMM_WORLD,
113                 &receive_requests[i]);
114         }
115
116         // Wait for all requests.
117         for (int i = 0; i < vectors.size(); i++) {
118             MPI_Wait(&receive_requests.at(i), &status);
119         }
120
121         // Kill all workers.
122         int kill_value = 1;
123         for (int i = 1; i < num_processes; i++) {
124             MPI_Send(&kill_value, 1, MPI_INT, i, kill_tag, MPI_COMM_WORLD);
125         }
126
127         double end = MPI_Wtime();
128         double total_time = end - begin;
129
130         cout << "Number processes: " << num_processes << endl;
131         cout << "Number vectors: " << number_vectors << endl;
132         cout << "Vector size: " << vector_size << endl;
133         cout << "Time sort (s): " << total_time << endl;
134     }
135     MPI_Finalize();
136
137     return 0;
138 }

```

Listing A.3: Bubble Sort MPI

Appendix B

Linear Regression Source Code

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 namespace dataset {
7 vector<int> get_vector(int vector_size) {
8     vector<int> v;
9     for (int i = 0; i < vector_size; i++) {
10         v.push_back(vector_size - i);
11     }
12     return v;
13 }
14
15 vector<vector<int>> get_dataset(int number_vectors, int vector_size) {
16     vector<vector<int>> vectors;
17     vector<int> v = get_vector(vector_size);
18     for (int i = 0; i < number_vectors; i++) {
19         vectors.push_back(v);
20     }
21     return vectors;
22 }
23 } // namespace dataset
```

Listing B.1: Dataset generator

```
1 #include "dataset-generator.cpp"
2 #include <chrono>
3 #include <cstdio>
4 #include <fstream>
5 #include <iostream>
6 #include <sstream>
7 #include <tuple>
8 #include <vector>
9
10 using namespace std;
11
12 vector<dataset::Point> load_dataset(unsigned long long int number_points) {
13     chrono::steady_clock::time_point begin = chrono::steady_clock::now();
14     vector<dataset::Point> points = dataset::get_dataset(number_points);
15     chrono::steady_clock::time_point end = chrono::steady_clock::now();
16     double total_time = chrono::duration<double>(end - begin).count();
17     cout << "Time load dataset (s): " << total_time << endl;
18     return points;
19 }
20
21 tuple<double, double, double> execute_lr(vector<dataset::Point> points) {
22     chrono::steady_clock::time_point begin = chrono::steady_clock::now();
23
24     unsigned long long int x_sum = 0;
25     unsigned long long int y_sum = 0;
26     unsigned long long int x_squared_sum = 0;
27     unsigned long long int xy_sum = 0;
28     int n = (int)points.size();
29
30     for (unsigned long long int i = 0; i < n; i++) {
31         int x_aux = points.at(i).x;
32         int y_aux = points.at(i).y;
33
34         x_sum += x_aux;
35         y_sum += y_aux;
```

```
37         x_squared_sum += x_aux * x_aux;
38         xy_sum += x_aux * y_aux;
39     }
40
41     chrono::steady_clock::time_point end = chrono::steady_clock::now();
42     double total_time = chrono::duration<double>(end - begin).count();
43
44     double slope = ((double)(n * xy_sum - x_sum * y_sum)) /
45         ((double)(n * x_squared_sum - x_sum * x_sum));
46     double intercept = ((double)(y_sum - slope * x_sum)) / n;
47
48     return make_tuple(total_time, slope, intercept);
49 }
50
51 int main(int argc, char **argv) {
52     unsigned long long int number_points = atoll(argv[1]);
53     vector<dataset::Point> points = load_dataset(number_points);
54     tuple<double, double, double> results = execute_lr(points);
55
56     double total_time = get<0>(results);
57     double slope = get<1>(results);
58     double intercept = get<2>(results);
59     cout << "Time linear regression (s): " << total_time << endl;
60     cout << "Slope: " << slope << endl;
61     cout << "Intercept: " << intercept << endl;
62
63     return 0;
64 }
```

Listing B.2: Linear Regression Sequential

```
1 #include "dataset-generator.cpp"
2 #include <chrono>
3 #include <cstdio>
4 #include <fstream>
5 #include <iostream>
6 #include <mpi.h>
7 #include <sstream>
8 #include <tuple>
9 #include <vector>
10
11 using namespace std;
12
13 // Store the results from each worker.
14 struct RegressionSubResults {
15     unsigned long long int x_sum;
16     unsigned long long int y_sum;
17     unsigned long long int x_squared_sum;
18     unsigned long long int xy_sum;
19 };
20
21 vector<dataset::Point> load_dataset(unsigned long long int number_points) {
22     double begin = MPI_Wtime();
23     vector<dataset::Point> points = dataset::get_dataset(number_points);
24     double end = MPI_Wtime();
25     double total_time = end - begin;
26     cout << "Time load dataset (s): " << total_time << endl;
27     return points;
28 }
29
30 // Perform linear regression on the subvector.
31 RegressionSubResults execute_lr(vector<dataset::Point> points) {
```

```

32 unsigned long long int x_sum = 0;
33 unsigned long long int y_sum = 0;
34 unsigned long long int x_squared_sum = 0;
35 unsigned long long int xy_sum = 0;
36 int n = (int)points.size();
37
38 for (unsigned long long int i = 0; i < n; i++) {
39     int x_aux = points.at(i).x;
40     int y_aux = points.at(i).y;
41
42     x_sum += x_aux;
43     y_sum += y_aux;
44
45     x_squared_sum += x_aux * x_aux;
46     xy_sum += x_aux * y_aux;
47 }
48
49 return {
50     .x_sum = x_sum,
51     .y_sum = y_sum,
52     .x_squared_sum = x_squared_sum,
53     .xy_sum = xy_sum,
54 };
55 }
56
57 int main(int argc, char **argv) {
58     unsigned long long int number_points = atoll(argv[1]);
59     unsigned long long int granularity = atoll(argv[2]);
60
61     int vector_tag = 1;
62     int kill_tag = 2;
63     int request_vector_tag = 3;
64
65     int number_grains = number_points / granularity;
66     MPI_Status status;
67     int my_rank;
68     int num_processes;
69
70     MPI_Init(&argc, &argv);
71     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
72     MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
73
74     if ((number_points % granularity) > 0) {
75         // This avoids the need to deal with the last elements of the array.
76         cout << "Error: granularity must be a multiple of the number of points."
77             << endl;
78         MPI_Abort(MPI_COMM_WORLD, -1);
79     }
80
81     // Commit Point struct to MPI.
82     MPI_Datatype MPI_POINT_TYPE;
83     int block_lengths_point[2] = {1, 1};
84     MPI_Aint displacements_point[2] = {offsetof(dataset::Point, x),
85                                       offsetof(dataset::Point, y)};
86     MPI_Datatype types_point[2] = {MPI_INT, MPI_INT};
87     MPI_Type_create_struct(2, block_lengths_point, displacements_point,
88                           types_point, &MPI_POINT_TYPE);
89     MPI_Type_commit(&MPI_POINT_TYPE);
90
91     // Commit RegressionSubResults struct to MPI.
92     MPI_Datatype MPI_REGRESSION_SUB_RESULTS_TYPE;
93     int block_lengths_regression_sub_results[4] = {1, 1, 1, 1};
94     MPI_Aint displacements_regression_sub_results[4] = {
95         offsetof(RegressionSubResults, x_sum),
96         offsetof(RegressionSubResults, y_sum),
97         offsetof(RegressionSubResults, x_squared_sum),
98         offsetof(RegressionSubResults, xy_sum)};
99     MPI_Datatype types_regression_sub_results[4] = {
100         MPI_LONG_LONG_INT, MPI_LONG_LONG_INT, MPI_LONG_LONG_INT,
101         MPI_LONG_LONG_INT};
102     MPI_Type_create_struct(4, block_lengths_regression_sub_results,
103                           displacements_regression_sub_results,
104                           types_regression_sub_results,
105                           &MPI_REGRESSION_SUB_RESULTS_TYPE);
106     MPI_Type_commit(&MPI_REGRESSION_SUB_RESULTS_TYPE);
107
108     if (my_rank != 0) {
109         int master = 0;
110         int ask_for_message = 1;
111         int kill_flag = 0;
112         while ((!kill_flag) {
113             if (ask_for_message) {
114                 // Will only send a new request when the last request was
115                 // already processed.
116                 MPI_Send(&ask_for_message, 1, MPI_INT, master,
117                        request_vector_tag, MPI_COMM_WORLD);
118                 ask_for_message = 0;
119             }
120             // Test whether the master submitted a new job.
121             int has_message = 0;
122             MPI_Iprobe(master, vector_tag, MPI_COMM_WORLD, &has_message,
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
&status);
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
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 (a): " << total_time << endl;
    cout << "Slope: " << slope << endl;
    cout << "Intercept: " << intercept << endl;
}
MPI_Finalize();
return 0;
}

```

Listing B.3: Linear Regression MPI