

# Bubble Sort and Linear Regression with MPI

Claudio Scheer<sup>1</sup> and Gabriell Araujo<sup>1</sup>

<sup>1</sup>Master'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 (2x24 cores) in the Cerrado 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'<sup>1</sup>.

## 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 results of the executions using the sequential (Listing A.2) and the MPI version (Listing A.3), with different numbers of slaves.

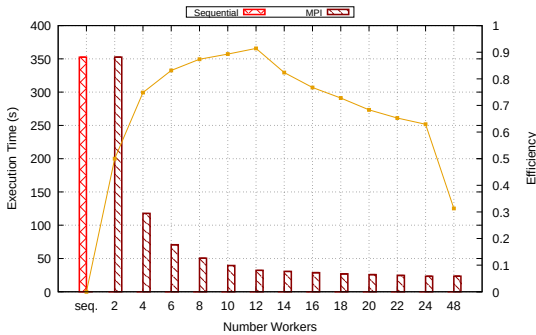


Figure 1: Execution Time x Efficiency

<sup>1</sup>What a horrible scenario!

As the number of processes increases, the execution time is shorter. However, the efficiency of the parallel execution grows slowly from 4 to 12 processes. Even so, the efficiency of the bubble sort with MPI reaches 90%. This indicates that, up to 12 processes, the bubble sort algorithm can exploit up to 90% of the expected speedup.

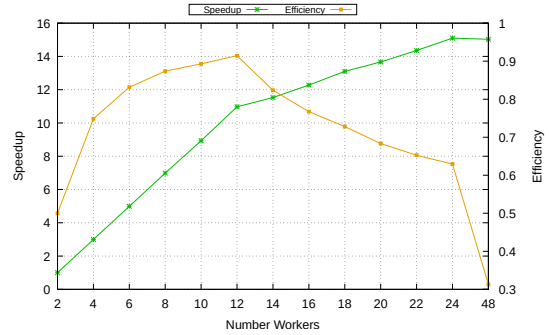


Figure 2: Speedup x Efficiency

Figure 2 shows that the speedup grows linearly for physical cores. However, when using hyper-threading, the speedup is only 31.31% effective when compared to the number of processes. Another fact is that when the cores are in different nodes, speedup and efficiency begins to decrease.

## 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, with granularity of 10000, 100000, 500000 and 1000000 points. Figure 3 shows the results for different number of processes and granularities, using the sequential (Listing B.2) and the MPI (Listing B.3) version.

In the configurations tested, lower granularity has a better performance. However, this may not be true for smaller granularities, as the number of messages sent will be higher. For the MPI version with two processes, the execution time was worse than the sequential version. It shows that the cost of sending

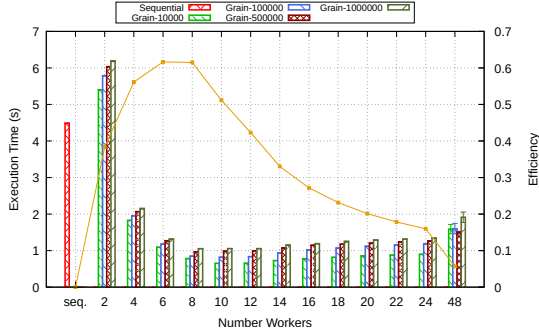


Figure 3: Execution Time x Efficiency

messages to each slave costs about 23% of the time. Figure 4 shows that, after 8 processes, the speedup stabilizes and efficiency drops. This fact shows that, regardless of the size of the input, the linear regression algorithm, using master-slave architecture, has a point where speedup and efficiency stop growing.

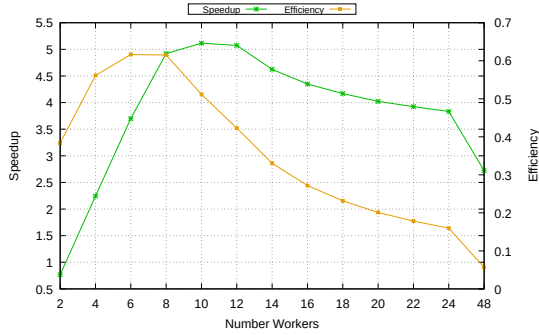


Figure 4: Speedup x Efficiency

## 0.4 Discussion

In linear regression, hyper-threading increased the execution time when compared to the approach using all physical cores and the efficiency was only 5.7%. For bubble sort, hyper-threading has a speedup, but not so efficient as using only the physical cores. Therefore, the characteristics of the problem influence the choice of whether or not to use hyper-threading. In addition, depending of the problem

addressed using MPI and master-slave architecture, more processes can decrease the speedup.

The main difference between the two problems is the speedup as the number of processes increases. Bubble sort will have a linear growth, while linear regression has a point where, even with more physical cores, the speedup stabilizes.

# 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 <cstdio>
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 <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 string get_hostname() {
14     std::ifstream file("/etc/hostname");
15     std::stringstream buffer;
16     buffer << file.rdbuf();
17     return buffer.str();
18 }
19
20 vector<vector<int>> load_dataset(int number_vectors, int vector_size) {
21     chrono::steady_clock::time_point begin = chrono::steady_clock::now();
22     vector<vector<int>> vectors =
23         dataset::get_dataset(number_vectors, vector_size);
24     chrono::steady_clock::time_point end = chrono::steady_clock::now();
25     double total_time =
26         chrono::duration_cast<chrono::duration<double>>(end - begin).count();
27     cout << "Time load dataset (s): " << total_time << endl;
28     return vectors;
29 }
30
31 vector<int> bubble_sort(vector<int> v) {
32     int n = v.size();
33     int c = 0;
34     int temp;
35     int swapped = 1;
```

```

36 while ((c < (n - 1)) & swapped) {
37     swapped = 0;
38     for (int d = 0; d < n - c - 1; d++)
39         if (v.at(d) > v.at(d + 1)) {
40             temp = v.at(d);
41             v.at(d) = v.at(d + 1);
42             v.at(d + 1) = temp;
43             swapped = 1;
44         }
45     c++;
46 }
47
48 return v;
49 }
50
51 int main(int argc, char **argv) {
52     int number_vectors = atoi(argv[1]);
53     int vector_size = atoi(argv[2]);
54
55     int vector_tag = 1;
56     int kill_tag = 2;
57     int request_vector_tag = 3;
58
59     MPI_Status status;
60     int my_rank;
61     int num_processes;
62
63     MPI_Init(&argc, &argv);
64     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
65     MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
66
67     cout << "Hostname (" << my_rank << "): " << gethostname() << endl;
68
69     if (my_rank != 0) {
70         int master = 0;
71         int ask_for_message = 1;
72         int kill_flag = 0;
73         while (!kill_flag) {
74             if (ask_for_message) {
75                 // Will only send a new request when the last request was
76                 // already processed.
77                 MPI_Send(&ask_for_message, 1, MPI_INT, master,
78                     request_vector_tag, MPI_COMM_WORLD);
79                 ask_for_message = 0;
80             }
81             // Test whether the master submitted a new job.
82             int has_message = 0;
83             MPI_Iprobe(master, vector_tag, MPI_COMM_WORLD, &has_message,
84                 &status);
85             if (has_message) {
86                 vector<int> v;
87                 v.resize(vector_size);
88                 MPI_Recv(&v[0], vector_size, MPI_INT, master, vector_tag,
89                     MPI_COMM_WORLD, &status);
90
91                 vector<int> v_sorted = bubble_sort(v);
92                 MPI_Send(&v_sorted[0], vector_size, MPI_INT, master, vector_tag,
93                     MPI_COMM_WORLD);
94
95                 ask_for_message = 1;
96             }
97             // Check for a 'suicide' request.
98             MPI_Iprobe(master, kill_tag, MPI_COMM_WORLD, &kill_flag, &status);
99         }
100     } else {
101         vector<vector<int>> vectors = load_dataset(number_vectors, vector_size);
102
103         double begin = MPI_Wtime();
104
105         // Store async requests received from workers.
106         vector<MPI_Request> receive_requests(number_vectors);
107         vector<vector<int>> ordered_vectors(number_vectors);
108
109         int worker_request = 0;
110         for (int i = 0; i < vectors.size(); i++) {
111             vector<int> v = vectors.at(i);
112             MPI_Recv(&worker_request, 1, MPI_INT, MPI_ANY_SOURCE,
113                 request_vector_tag, MPI_COMM_WORLD, &status);
114             // Send the vector to the worker
115             MPI_Send(&v[0], vector_size, MPI_INT, status.MPI_SOURCE, vector_tag,
116                 MPI_COMM_WORLD);
117
118             ordered_vectors[i].resize(vector_size);
119             MPI_Irecv(&ordered_vectors[i][0], vector_size, MPI_INT,
120                 status.MPI_SOURCE, vector_tag, MPI_COMM_WORLD,
121                 &receive_requests[i]);
122         }
123
124         // Wait for all requests.
125         for (int i = 0; i < vectors.size(); i++) {
126             MPI_Wait(&receive_requests.at(i), &status);
127         }
128
129         // Kill all workers.
130         int kill_value = 1;
131         for (int i = 1; i < num_processes; i++) {
132             MPI_Send(&kill_value, 1, MPI_INT, i, kill_tag, MPI_COMM_WORLD);
133         }
134
135         double end = MPI_Wtime();
136         double total_time = end - begin;
137
138         cout << "Number processes: " << num_processes << endl;
139         cout << "Number vectors: " << number_vectors << endl;
140         cout << "Vector size: " << vector_size << endl;
141         cout << "Time sort (s): " << total_time << endl;
142     }
143     MPI_Finalize();
144     return 0;
145 }

```

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;
36
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 string get_hostname() {
22     std::ifstream file("/etc/hostname");
23     std::stringstream buffer;
24     buffer << file.rdbuf();
25     return buffer.str();
26 }
27
28 vector<dataset::Point> load_dataset(unsigned long long int number_points) {
29     double begin = MPI_Wtime();
30     vector<dataset::Point> points = dataset::get_dataset(number_points);
31     double end = MPI_Wtime();
```

```

32     double total_time = end - begin;
33     cout << "Time load dataset (s): " << total_time << endl;
34     return points;
35 }
36
37 // Perform linear regression on the subvector.
38 RegressionSubResults execute_lr(vector<dataset::Point> points) {
39     unsigned long long int x_sum = 0;
40     unsigned long long int y_sum = 0;
41     unsigned long long int x_squared_sum = 0;
42     unsigned long long int xy_sum = 0;
43     int n = (int)points.size();
44
45     for (unsigned long long int i = 0; i < n; i++) {
46         int x_aux = points.at(i).x;
47         int y_aux = points.at(i).y;
48
49         x_sum += x_aux;
50         y_sum += y_aux;
51
52         x_squared_sum += x_aux * x_aux;
53         xy_sum += x_aux * y_aux;
54     }
55
56     return {
57         .x_sum = x_sum,
58         .y_sum = y_sum,
59         .x_squared_sum = x_squared_sum,
60         .xy_sum = xy_sum,
61     };
62 }
63
64 int main(int argc, char **argv) {
65     unsigned long long int number_points = atoll(argv[1]);
66     unsigned long long int granularity = atoll(argv[2]);
67
68     int vector_tag = 1;
69     int kill_tag = 2;
70     int request_vector_tag = 3;
71
72     int number_grains = number_points / granularity;
73     MPI_Status status;
74     int my_rank;
75     int num_processes;
76
77     MPI_Init(&argc, &argv);
78     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
79     MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
80
81     cout << "Hostname (" << my_rank << ") : " << gethostname() << endl;
82
83     if ((number_points % granularity) > 0) {
84         // This avoids the need to deal with the last elements of the array.
85         cout << "Error: granularity must be a multiple of the number of points."
86             << endl;
87         MPI_Abort(MPI_COMM_WORLD, -1);
88     }
89
90     // Commit Point struct to MPI.
91     MPI_Datatype MPI_POINT_TYPE;
92     int block_lengths_point[2] = {1, 1};
93     MPI_Aint displacements_point[2] = {offsetof(dataset::Point, x),
94                                         offsetof(dataset::Point, y)};
95     MPI_Datatype types_point[2] = {MPI_INT, MPI_INT};
96     MPI_Type_create_struct(2, block_lengths_point, displacements_point,
97                           types_point, &MPI_POINT_TYPE);
98     MPI_Type_commit(&MPI_POINT_TYPE);
99
100    // Commit RegressionSubResults struct to MPI.
101    MPI_Datatype MPI_REGRESSION_SUB_RESULTS_TYPE;
102    int block_lengths_regression_sub_results[4] = {1, 1, 1, 1};
103    MPI_Aint displacements_regression_sub_results[4] = {
104        offsetof(RegressionSubResults, x_sum),
105        offsetof(RegressionSubResults, y_sum),
106        offsetof(RegressionSubResults, x_squared_sum),
107        offsetof(RegressionSubResults, xy_sum)};
108    MPI_Datatype types_regression_sub_results[4] = {
109        MPI_LONG_LONG_INT, MPI_LONG_LONG_INT, MPI_LONG_LONG_INT,
110        MPI_LONG_LONG_INT};
111    MPI_Type_create_struct(4, block_lengths_regression_sub_results,
112                          displacements_regression_sub_results,
113                          types_regression_sub_results,
114                          &MPI_REGRESSION_SUB_RESULTS_TYPE);
115    MPI_Type_commit(&MPI_REGRESSION_SUB_RESULTS_TYPE);
116
117    if (my_rank != 0) {
118        int master = 0;
119        int ask_for_message = 1;
120        int kill_flag = 0;
121        while (!kill_flag) {
122            if (ask_for_message) {
123
124                // Will only send a new request when the last request was
125                // already processed.
126                MPI_Send(&ask_for_message, 1, MPI_INT, master,
127                        request_vector_tag, MPI_COMM_WORLD);
128                ask_for_message = 0;
129            }
130            // Test whether the master submitted a new job.
131            int has_message = 0;
132            MPI_Iprobe(master, vector_tag, MPI_COMM_WORLD, &has_message,
133                      &status);
134            if (has_message) {
135                vector<dataset::Point> points;
136                points.resize(granularity);
137                MPI_Recv(&points[0], granularity, MPI_POINT_TYPE, master,
138                        vector_tag, MPI_COMM_WORLD, &status);
139
140                RegressionSubResults sub_results = execute_lr(points);
141                MPI_Send(&sub_results, 1, MPI_REGRESSION_SUB_RESULTS_TYPE,
142                        master, vector_tag, MPI_COMM_WORLD);
143
144                ask_for_message = 1;
145            }
146            // Check for a 'suicide' request.
147            MPI_Iprobe(master, kill_tag, MPI_COMM_WORLD, &kill_flag, &status);
148        }
149    } else {
150        vector<dataset::Point> points = load_dataset(number_points);
151
152        double begin = MPI_Wtime();
153
154        // Store async requests received from workers.
155        vector<MPI_Request> receive_requests(number_grains);
156        vector<RegressionSubResults> regression_sub_results(number_grains);
157
158        int grain = 0;
159        int worker_request = 0;
160        while (number_points > (grain * granularity)) {
161            MPI_Recv(&worker_request, 1, MPI_INT, MPI_ANY_SOURCE,
162                    request_vector_tag, MPI_COMM_WORLD, &status);
163            // Send the next elements from the dataset to the worker.
164            MPI_Send(&points[(grain * granularity)], granularity,
165                    MPI_POINT_TYPE, status.MPI_SOURCE, vector_tag,
166                    MPI_COMM_WORLD);
167            MPI_Irecv(&regression_sub_results[grain], 1,
168                    MPI_REGRESSION_SUB_RESULTS_TYPE, status.MPI_SOURCE,
169                    vector_tag, MPI_COMM_WORLD, &receive_requests[grain]);
170            grain++;
171        }
172
173        RegressionSubResults results = {
174            .x_sum = 0,
175            .y_sum = 0,
176            .x_squared_sum = 0,
177            .xy_sum = 0,
178        };
179
180        // Collect the results of all workers.
181        for (int i = 0; i < number_grains; i++) {
182            MPI_Wait(&receive_requests.at(i), &status);
183            RegressionSubResults sub_results = regression_sub_results.at(i);
184            results.x_sum += sub_results.x_sum;
185            results.y_sum += sub_results.y_sum;
186            results.x_squared_sum += sub_results.x_squared_sum;
187            results.xy_sum += sub_results.xy_sum;
188        }
189
190        // Kill all workers.
191        int kill_value = 1;
192        for (int i = 1; i < num_processes; i++) {
193            MPI_Send(&kill_value, 1, MPI_INT, i, kill_tag, MPI_COMM_WORLD);
194        }
195
196        double end = MPI_Wtime();
197        double total_time = end - begin;
198
199        double slope = ((double)(number_points * results.xy_sum -
200                                results.x_sum * results.y_sum)) /
201            ((double)(number_points * results.x_squared_sum -
202                                results.x_sum * results.x_sum));
203
204        double intercept =
205            ((double)(results.y_sum - slope * results.x_sum)) / number_points;
206        cout << "Time linear regression (s): " << total_time << endl;
207        cout << "Slope: " << slope << endl;
208        cout << "Intercept: " << intercept << endl;
209    }
210    MPI_Finalize();
211    return 0;

```

Listing B.3: Linear Regression MPI