

Bubble Sort using Domain Decomposition with MPI

Claudio Scheer, Gabriell Araujo

General Setup

We ran our *batch job* on two nodes (2x12 cores, 2x24 when considering hyper-threading) in the Cerrado cluster. All experiments were executed three times and then the average execution time and the standard deviation were calculated. Efficiency and speedup were based on the execution time reported by the sequential execution of the bubble sort algorithm.

For the implementation using MPI, we used the domain decomposition philosophy. In short, each process sorts $1/n$ of the vector using the bubble sort algorithm and shares its lowest values with the left neighbor. The piece of the vector shared is interleaved with the vector held by the left neighbor, and the same process is repeated until the vector is sorted. In this report, we discuss three optimization that we perform in this workflow.

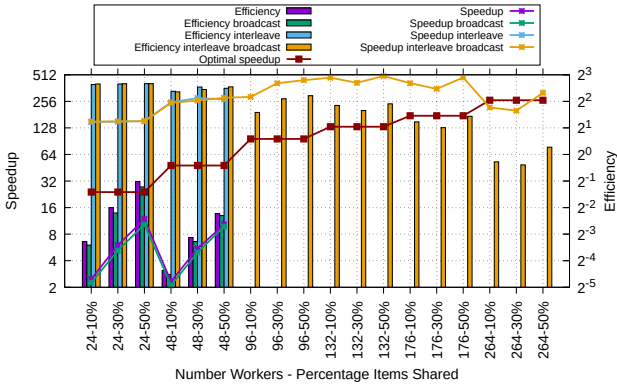


Figure 1: Speedup x Efficiency

Bubble Sort without optimizations

The bubble sort problem addressed here consists of sorting one vector with 1000032 integers. This vector is divided among the processes. Therefore, each process have $n/\text{processes}$ integers. As we are sorting a vector in descending order, each process can generate its own vector, without the need to create the entire vector and share it between the processes.

The first phase of the bubble sort algorithm using domain decomposition structure is the application of the bubble sort algorithm over the part of the vector held by each process. After that, each processes communicates with all other processes to test whether they are sorted or not with their neighboring processes. Otherwise, the processes send their lowest numbers to the process on left and receive the highest values from the process on the right. This steps is repeated until the vector is sorted.

According to the results seen in Figure 1, the speedup of this implementation is deeply related to the percentage of numbers that are shared between the processes. It is important to note that we cannot shared more than 50% of the vector between processes, since processes other than 0 and $\text{processes} - 1$ will send and receive numbers from the left and right.

Broadcast optimization

The first optimization was to reduce the amount of broadcast messages sent by processes to check whether they are all sorted or not when compared with the other processes. Therefore, as soon as a broadcast message returns that a process is not sorted in relation to the left neighbor, we stop sending and receiving broadcast messages.

As shown in Figure 1, the results using this optimization, compared to results without optimization, were in the range of the standard deviation. Even when the percentage of numbers shared between the processes were higher, which results in faster convergence, the reduction in broadcast messages was not significant. TALK ABOUT THE STRESS TEST.

Interleave instead of bubble sort

The second optimization was focused on reducing the use of the bubble sort algorithm as much as possible, since this algorithm has a time complexity of $O(n)$. Thus, as we can interleave the vector with the part of the vector shared between the processes, it is possible to execute the bubble sort algorithm only once in each process. To use interleaving instead of bubble sort, we first interleave the vector with the percentage of numbers received from right, ignoring the percentage of numbers received from the left. Finally, we interleave the ignored part with the entire vector. It is not possible to call just once the interleave method, as we can end up with three vector pieces in a single process.

This optimization lead to a speedup higher than the optimal speedup. As the interleave algorithm has a linear time complexity, the percentage of items shared between the processes does not have a high impact on speedup or efficiency when using all physical cores or hyper-threading.

Broadcast and interleave

We also tested using both optimization discussed above at the same time. The results showed that this can lead to almost the same results as using just the interleaving optimization. However, when using more process than hyper-threading limit, the results shows that sharing a larger percentage of number is better. Despite this, there is a loss of efficiency and, at some point, a lost of speedup. We believe that this is due to the number of broadcast messages that must be used to synchronize all of these processes.

Discussion

Finally, the broadcast messages, even when widely required, have no significant impact on the domain decomposition performance with MPI. The main bottleneck in this process is the bubble sort algorithm.

Comparing the domain decomposition, with its better optimization, with the divide and conquer approach shows that divide and conquer has a much better speedup and efficiency when using more processes than hyper-threading.

Bubble Sort Source Code

Listing 1: Dataset generator

```
1  #include <stdio.h>
2
3  int *get_vector(int vector_size, int *v) {
4      for (int i = 0; i < vector_size; i++) {
5          v[i] = vector_size - i;
6      }
7      return v;
8  }
9
10 int *get_vector_offset(int vector_size, int subvector_size, int *v,
11                        int offset) {
12     int init = subvector_size * offset;
13     int end = init + subvector_size;
14     for (int i = init; i < end; i++) {
15         v[i - init] = vector_size - i;
16     }
17     return v;
18 }
```

Listing 2: Bubble Sort Sequential

```
1  #include "dataset-generator.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/time.h>
5
6  float time_difference_msec(struct timeval t0, struct timeval t1) {
7      return (t1.tv_sec - t0.tv_sec) * 1000.0f +
8             (t1.tv_usec - t0.tv_usec) / 1000.0f;
9  }
10
11 int is_vector_sorted(int *vector, int vector_size) {
12     int last_element = -1;
13     for (int i = 0; i < vector_size; i++) {
14         if (last_element > vector[i]) {
15             return 0;
16         }
17         last_element = vector[i];
18     }
19     return 1;
20 }
21
22 int *bubble_sort(int vector_size, int *vector_unsorted) {
23     int swapped = 1;
24
25     int i = 0;
26     while ((i < (vector_size - 1)) & swapped) {
27         swapped = 0;
28         for (int j = 0; j < vector_size - i - 1; j++)
29             if (vector_unsorted[j] > vector_unsorted[j + 1]) {
30                 int temp = vector_unsorted[j];
31                 vector_unsorted[j] = vector_unsorted[j + 1];
32                 vector_unsorted[j + 1] = temp;
33                 swapped = 1;
34             }
35         i++;
36     }
37
38     return vector_unsorted;
39 }
40
41 int main(int argc, char **argv) {
42     int vector_size = atoi(argv[1]);
43
44     int vector_unsorted[vector_size];
45     get_vector(vector_size, vector_unsorted);
46
47     struct timeval t0;
```

```

48     struct timeval t1;
49
50     gettimeofday(&t0, 0);
51     int *vector_sorted = bubble_sort(vector_size, vector_unsorted);
52     gettimeofday(&t1, 0);
53
54     float total_time = time_difference_msec(t0, t1);
55
56     printf("Vector sorted: %d\n", is_vector_sorted(vector_sorted, vector_size));
57     printf("Vector size: %d\n", vector_size);
58     printf("Time sort (ms): %f\n", total_time);
59
60     return 0;
61 }

```

Listing 3: Bubble Sort MPI

```

1  #include "dataset-generator.h"
2  #include <mpi.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/time.h>
6  #define DEBUG 0
7  #define OPTIMIZE_BROADCAST 1
8  #define BUBBLE_SORT_ONLY_ONCE 1
9
10 void interleave_vector(int *vector, int a_init_index, int a_end_index,
11                       int b_init_index, int b_end_index, int *result) {
12     if (a_end_index - a_init_index > b_end_index - b_init_index) {
13         int temp = a_init_index;
14         a_init_index = b_init_index;
15         b_init_index = temp;
16         temp = a_end_index;
17         a_end_index = b_end_index;
18         b_end_index = temp;
19     }
20
21     int i = 0, a = a_init_index, b = b_init_index;
22     while (a <= a_end_index & b <= b_end_index) {
23         if (a <= a_end_index && (vector[a] < vector[b] & b > b_end_index)) {
24             result[i] = vector[a];
25             a++;
26         } else {
27             result[i] = vector[b];
28             b++;
29         }
30         i++;
31     }
32 }
33
34 void bubble_sort(int vector_size, int *vector_unsorted) {
35     int swapped = 1;
36
37     int i = 0;
38     while ((i < (vector_size - 1)) & swapped) {
39         swapped = 0;
40         for (int j = 0; j < vector_size - i - 1; j++)
41             if (vector_unsorted[j] > vector_unsorted[j + 1]) {
42                 int temp = vector_unsorted[j];
43                 vector_unsorted[j] = vector_unsorted[j + 1];
44                 vector_unsorted[j + 1] = temp;
45                 swapped = 1;
46             }
47         i++;
48     }
49 }
50
51 int is_vector_sorted(int *vector, int vector_size) {
52     int last_element = -1;
53     for (int i = 0; i < vector_size; i++) {
54         if (last_element > vector[i]) {
55             return 0;

```

```

56     }
57     last_element = vector[i];
58 }
59 return 1;
60 }
61
62 int main(int argc, char **argv) {
63     int vector_size = atoi(argv[1]);
64     float percentage_items_exchange = atof(argv[2]);
65
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 (vector_size % num_processes) {
75         printf("Error: vector size must be a multiple of the number of "
76             "processes.\n");
77         MPI_Abort(MPI_COMM_WORLD, -1);
78     }
79
80     double t0;
81     double t1;
82     t0 = MPI_Wtime();
83
84     int subvector_size = vector_size / num_processes;
85     int number_items_shared = subvector_size * percentage_items_exchange;
86     int subvector[subvector_size + number_items_shared];
87     get_vector_offset(vector_size, subvector_size, subvector, my_rank);
88
89     // Avoid allocating a new vector in each iteration.
90     int process_status[num_processes];
91     int interleaved_shared_vector_pre_defined[number_items_shared * 2];
92 #if BUBBLE_SORT_ONLY_ONCE == 1
93     int interleaved_subvector_pre_defined[subvector_size];
94     int bubble_sort_executed = 0;
95 #endif
96
97 #if DEBUG == 1
98     int loop_index = 0;
99 #endif
100
101     int done = 0;
102     while (!done) {
103         // -----FIRST PHASE-----
104 #if BUBBLE_SORT_ONLY_ONCE == 1
105         if (!bubble_sort_executed) {
106             bubble_sort(subvector_size, subvector);
107             bubble_sort_executed = 1;
108         } else {
109             interleave_vector(subvector, number_items_shared,
110                             subvector_size - number_items_shared - 1,
111                             subvector_size - number_items_shared,
112                             subvector_size - 1,
113                             interleaved_subvector_pre_defined);
114             for (int i = 0; i < subvector_size - number_items_shared; i++) {
115                 subvector[i + number_items_shared] =
116                     interleaved_subvector_pre_defined[i];
117             }
118
119             interleave_vector(subvector, 0, number_items_shared - 1,
120                             number_items_shared, subvector_size - 1,
121                             interleaved_subvector_pre_defined);
122             for (int i = 0; i < subvector_size; i++) {
123                 subvector[i] = interleaved_subvector_pre_defined[i];
124             }
125         }
126 #else

```

```

127     bubble_sort(subvector_size, subvector);
128 #endif
129 // -----FIRST PHASE-----
130
131 // -----SECOND PHASE-----
132 if (my_rank < num_processes - 1) {
133     MPI_Send(&subvector[subvector_size - 1], 1, MPI_INT, my_rank + 1, 0,
134             MPI_COMM_WORLD);
135 }
136 if (my_rank > 0) {
137     int largest_number_left;
138     MPI_Recv(&largest_number_left, 1, MPI_INT, my_rank - 1, 0,
139             MPI_COMM_WORLD, &status);
140     process_status[my_rank] = subvector[0] > largest_number_left;
141 } else {
142     process_status[my_rank] = 1;
143 }
144
145 for (int i = 0; i < num_processes; i++) {
146     MPI_Bcast(&process_status[i], 1, MPI_INT, i, MPI_COMM_WORLD);
147 #if OPTIMIZE_BROADCAST == 1
148     if (!process_status[i]) {
149         break;
150     }
151 #endif
152 }
153
154 int vector_sorted = 1;
155 for (int i = 0; i < num_processes; i++) {
156     vector_sorted &= process_status[i];
157     if (!vector_sorted) {
158         break;
159     }
160 }
161 if (vector_sorted) {
162     done = 1;
163     break;
164 }
165 // -----SECOND PHASE-----
166
167 // -----THIRD PHASE-----
168 if (my_rank > 0) {
169     MPI_Send(&subvector[0], number_items_shared, MPI_INT, my_rank - 1,
170             0, MPI_COMM_WORLD);
171 }
172 if (my_rank < num_processes - 1) {
173     MPI_Recv(&subvector[subvector_size], number_items_shared, MPI_INT,
174             my_rank + 1, 0, MPI_COMM_WORLD, &status);
175
176     interleave_vector(subvector, subvector_size - number_items_shared,
177                       subvector_size - 1, subvector_size,
178                       subvector_size + number_items_shared - 1,
179                       interleaved_shared_vector_pre_defined);
180
181     MPI_Send(
182         &interleaved_shared_vector_pre_defined[number_items_shared],
183         number_items_shared, MPI_INT, my_rank + 1, 0, MPI_COMM_WORLD);
184
185     for (int i = 0; i < number_items_shared; i++) {
186         subvector[subvector_size - number_items_shared + i] =
187             interleaved_shared_vector_pre_defined[i];
188     }
189 }
190 if (my_rank > 0) {
191     MPI_Recv(&subvector[0], number_items_shared, MPI_INT, my_rank - 1,
192             0, MPI_COMM_WORLD, &status);
193 }
194 // -----THIRD PHASE-----
195 #if DEBUG == 1
196     if (loop_index == 0) {
197         for (int i = 0; i < subvector_size; i++) {

```

```

198         printf("%d ", subvector[i]);
199     }
200     printf("----- %d", my_rank);
201     printf("\n");
202     fflush(stdout);
203 }
204 loop_index++;
205 #endif
206 }
207
208 t1 = MPI_Wtime();
209 double total_time = (t1 - t0) * 1000;
210
211 #if DEBUG == 1
212     if (my_rank > 0) {
213         MPI_Send(&subvector[0], subvector_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
214     } else {
215         int final_vector[vector_size];
216         for (int i = 0; i < subvector_size; i++) {
217             final_vector[i] = subvector[i];
218         }
219
220         for (int i = 1; i < num_processes; i++) {
221             int subvector_received[subvector_size];
222             MPI_Recv(&subvector_received[0], subvector_size, MPI_INT, i, 0,
223                     MPI_COMM_WORLD, &status);
224
225             for (int j = 0; j < subvector_size; j++) {
226                 final_vector[subvector_size * i + j] = subvector_received[j];
227             }
228         }
229
230         printf("Final vector:\n");
231         for (int i = 0; i < vector_size; i++) {
232             printf("%d ", final_vector[i]);
233         }
234         printf("\n");
235
236         printf("DEBUG: %d\n", DEBUG);
237         printf("OPTIMIZE_BROADCAST: %d\n", OPTIMIZE_BROADCAST);
238         printf("BUBBLE_SORT_ONLY_ONCE: %d\n", BUBBLE_SORT_ONLY_ONCE);
239         printf("Vector sorted: %d\n",
240               is_vector_sorted(subvector, subvector_size));
241         printf("Vector size: %d\n", vector_size);
242         printf("Time sort (ms): %f\n", total_time);
243     }
244 #else
245     if (my_rank == 0) {
246         printf("DEBUG: %d\n", DEBUG);
247         printf("OPTIMIZE_BROADCAST: %d\n", OPTIMIZE_BROADCAST);
248         printf("BUBBLE_SORT_ONLY_ONCE: %d\n", BUBBLE_SORT_ONLY_ONCE);
249         printf("Vector sorted: %d\n",
250               is_vector_sorted(subvector, subvector_size));
251         printf("Vector size: %d\n", vector_size);
252         printf("Time sort (ms): %f\n", total_time);
253     }
254 #endif
255
256 MPI_Finalize();
257
258 return 0;
259 }

```