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.

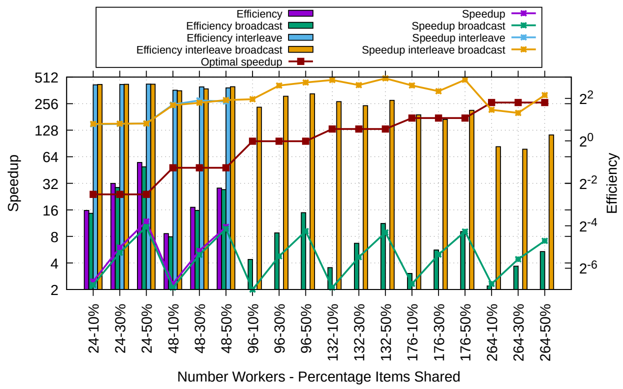

Figure 1: Speedup x Efficiency

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 $n/processes$ 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 $processes - 1$ 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. When using up to 264 processes, broadcast messages have an impact of about $\approx 66\%$, but we can still see that broadcast messages are not the main bottleneck.

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 $O(n^2)$. 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 only 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 the two optimizations 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 higher percentage of numbers 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

```c
#include <stdio.h>

int *get_vector(int vector_size, int *v) {
    for (int i = 0; i < vector_size; i++) {
        v[i] = vector_size - i;
    }
    return v;
}

int *get_vector_offset(int vector_size, int subvector_size, int *v,
                       int offset) {
    int init = subvector_size * offset;
    int end = init + subvector_size;
    for (int i = init; i < end; i++) {
        v[i - init] = vector_size - i;
    }
    return v;
}
```

Listing 2: Bubble Sort Sequential

```c
#include "dataset-generator.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

float time_difference_msec(struct timeval t0, struct timeval t1) {
    return (t1.tv_sec - t0.tv_sec) * 1000.0f +
           (t1.tv_usec - t0.tv_usec) / 1000.0f;
}

int is_vector_sorted(int *vector, int vector_size) {
    int last_element = -1;
    for (int i = 0; i < vector_size; i++) {
        if (last_element > vector[i]) {
            return 0;
        }
        last_element = vector[i];
    }
    return 1;
}

int *bubble_sort(int vector_size, int *vector_unsorted) {
    int swapped = 1;

    int i = 0;
    while ((i < (vector_size - 1)) & swapped) {
        swapped = 0;
        for (int j = 0; j < vector_size - i - 1; j++)
            if (vector_unsorted[j] > vector_unsorted[j + 1]) {
                int temp = vector_unsorted[j];
                vector_unsorted[j] = vector_unsorted[j + 1];
                vector_unsorted[j + 1] = temp;
                swapped = 1;
            }
        i++;
    }

    return vector_unsorted;
}

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

    int vector_unsorted[vector_size];
    get_vector(vector_size, vector_unsorted);

    struct timeval t0;
```

```c
    struct timeval t1;

    gettimeofday(&t0, 0);
    int *vector_sorted = bubble_sort(vector_size, vector_unsorted);
    gettimeofday(&t1, 0);

    float total_time = time_difference_msec(t0, t1);

    printf("Vector sorted: %d\n", is_vector_sorted(vector_sorted, vector_size));
    printf("Vector size: %d\n", vector_size);
    printf("Time sort (ms): %f\n", total_time);

    return 0;
}
```

Listing 3: Bubble Sort MPI

```c
#include "dataset-generator.h"
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define DEBUG 0
#define OPTIMIZE_BROADCAST 1
#define BUBBLE_SORT_ONLY_ONCE 1

void interleave_vector(int *vector, int a_init_index, int a_end_index,
                       int b_init_index, int b_end_index, int *result) {
    if (a_end_index - a_init_index > b_end_index - b_init_index) {
        int temp = a_init_index;
        a_init_index = b_init_index;
        b_init_index = temp;
        temp = a_end_index;
        a_end_index = b_end_index;
        b_end_index = temp;
    }

    int i = 0, a = a_init_index, b = b_init_index;
    while (a <= a_end_index  b <= b_end_index) {
        if (a <= a_end_index && (vector[a] < vector[b]  b > b_end_index)) {
            result[i] = vector[a];
            a++;
        } else {
            result[i] = vector[b];
            b++;
        }
        i++;
    }
}

void bubble_sort(int vector_size, int *vector_unsorted) {
    int swapped = 1;

    int i = 0;
    while ((i < (vector_size - 1)) & swapped) {
        swapped = 0;
        for (int j = 0; j < vector_size - i - 1; j++)
            if (vector_unsorted[j] > vector_unsorted[j + 1]) {
                int temp = vector_unsorted[j];
                vector_unsorted[j] = vector_unsorted[j + 1];
                vector_unsorted[j + 1] = temp;
                swapped = 1;
            }
        i++;
    }
}

int is_vector_sorted(int *vector, int vector_size) {
    int last_element = -1;
    for (int i = 0; i < vector_size; i++) {
        if (last_element > vector[i]) {
            return 0;
```

```c
        }
        last_element = vector[i];
    }
    return 1;
}

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

    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 (vector_size % num_processes) {
        printf("Error: vector size must be a multiple of the number of "
               "processes.\n");
        MPI_Abort(MPI_COMM_WORLD, -1);
    }

    double t0;
    double t1;
    t0 = MPI_Wtime();

    int subvector_size = vector_size / num_processes;
    int number_items_shared = subvector_size * percentage_items_exchange;
    int subvector[subvector_size + number_items_shared];
    get_vector_offset(vector_size, subvector_size, subvector, my_rank);

    // Avoid allocating a new vector in each iteration.
    int process_status[num_processes];
    int interleaved_shared_vector_pre_defined[number_items_shared * 2];
#if BUBBLE_SORT_ONLY_ONCE == 1
    int interleaved_subvector_pre_defined[subvector_size];
    int bubble_sort_executed = 0;
#endif

#if DEBUG == 1
    int loop_index = 0;
#endif

    int done = 0;
    while (!done) {
        // --------------------FIRST PHASE--------------------
#if BUBBLE_SORT_ONLY_ONCE == 1
        if (!bubble_sort_executed) {
            bubble_sort(subvector_size, subvector);
            bubble_sort_executed = 1;
        } else {
            interleave_vector(subvector, number_items_shared,
                              subvector_size - number_items_shared - 1,
                              subvector_size - number_items_shared,
                              subvector_size - 1,
                              interleaved_subvector_pre_defined);
            for (int i = 0; i < subvector_size - number_items_shared; i++) {
                subvector[i + number_items_shared] =
                    interleaved_subvector_pre_defined[i];
            }

            interleave_vector(subvector, 0, number_items_shared - 1,
                              number_items_shared, subvector_size - 1,
                              interleaved_subvector_pre_defined);
            for (int i = 0; i < subvector_size; i++) {
                subvector[i] = interleaved_subvector_pre_defined[i];
            }
        }
#else
```

```
        bubble_sort(subvector_size, subvector);
#endif
        // --------------------FIRST PHASE--------------------

        // --------------------SECOND PHASE--------------------
        if (my_rank < num_processes - 1) {
            MPI_Send(&subvector[subvector_size - 1], 1, MPI_INT, my_rank + 1, 0,
                     MPI_COMM_WORLD);
        }
        if (my_rank > 0) {
            int largest_number_left;
            MPI_Recv(&largest_number_left, 1, MPI_INT, my_rank - 1, 0,
                     MPI_COMM_WORLD, &status);
            process_status[my_rank] = subvector[0] > largest_number_left;
        } else {
            process_status[my_rank] = 1;
        }

        for (int i = 0; i < num_processes; i++) {
            MPI_Bcast(&process_status[i], 1, MPI_INT, i, MPI_COMM_WORLD);
#if OPTIMIZE_BROADCAST == 1
            if (!process_status[i]) {
                break;
            }
#endif
        }

        int vector_sorted = 1;
        for (int i = 0; i < num_processes; i++) {
            vector_sorted &= process_status[i];
            if (!vector_sorted) {
                break;
            }
        }
        if (vector_sorted) {
            done = 1;
            break;
        }
        // --------------------SECOND PHASE--------------------

        // --------------------THIRD PHASE--------------------
        if (my_rank > 0) {
            MPI_Send(&subvector[0], number_items_shared, MPI_INT, my_rank - 1,
                     0, MPI_COMM_WORLD);
        }
        if (my_rank < num_processes - 1) {
            MPI_Recv(&subvector[subvector_size], number_items_shared, MPI_INT,
                     my_rank + 1, 0, MPI_COMM_WORLD, &status);

            interleave_vector(subvector, subvector_size - number_items_shared,
                              subvector_size - 1, subvector_size,
                              subvector_size + number_items_shared - 1,
                              interleaved_shared_vector_pre_defined);

            MPI_Send(
                &interleaved_shared_vector_pre_defined[number_items_shared],
                number_items_shared, MPI_INT, my_rank + 1, 0, MPI_COMM_WORLD);

            for (int i = 0; i < number_items_shared; i++) {
                subvector[subvector_size - number_items_shared + i] =
                    interleaved_shared_vector_pre_defined[i];
            }
        }
        if (my_rank > 0) {
            MPI_Recv(&subvector[0], number_items_shared, MPI_INT, my_rank - 1,
                     0, MPI_COMM_WORLD, &status);
        }
        // --------------------THIRD PHASE--------------------
#if DEBUG == 1
        if (loop_index == 0) {
            for (int i = 0; i < subvector_size; i++) {
```

```c
                printf("%d ", subvector[i]);
            }
            printf("----- %d", my_rank);
            printf("\n");
            fflush(stdout);
        }
        loop_index++;
#endif
    }

    t1 = MPI_Wtime();
    double total_time = (t1 - t0) * 1000;

#if DEBUG == 1
    if (my_rank > 0) {
        MPI_Send(&subvector[0], subvector_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
    } else {
        int final_vector[vector_size];
        for (int i = 0; i < subvector_size; i++) {
            final_vector[i] = subvector[i];
        }

        for (int i = 1; i < num_processes; i++) {
            int subvector_received[subvector_size];
            MPI_Recv(&subvector_received[0], subvector_size, MPI_INT, i, 0,
                     MPI_COMM_WORLD, &status);

            for (int j = 0; j < subvector_size; j++) {
                final_vector[subvector_size * i + j] = subvector_received[j];
            }
        }

        printf("Final vector:\n");
        for (int i = 0; i < vector_size; i++) {
            printf("%d ", final_vector[i]);
        }
        printf("\n");

        printf("DEBUG: %d\n", DEBUG);
        printf("OPTIMIZE_BROADCAST: %d\n", OPTIMIZE_BROADCAST);
        printf("BUBBLE_SORT_ONLY_ONCE: %d\n", BUBBLE_SORT_ONLY_ONCE);
        printf("Vector sorted: %d\n",
               is_vector_sorted(subvector, subvector_size));
        printf("Vector size: %d\n", vector_size);
        printf("Time sort (ms): %f\n", total_time);
    }
#else
    if (my_rank == 0) {
        printf("DEBUG: %d\n", DEBUG);
        printf("OPTIMIZE_BROADCAST: %d\n", OPTIMIZE_BROADCAST);
        printf("BUBBLE_SORT_ONLY_ONCE: %d\n", BUBBLE_SORT_ONLY_ONCE);
        printf("Vector sorted: %d\n",
               is_vector_sorted(subvector, subvector_size));
        printf("Vector size: %d\n", vector_size);
        printf("Time sort (ms): %f\n", total_time);
    }
#endif

    MPI_Finalize();

    return 0;
}
```