Bubble Sort using Divide and Conquer with MPI

Claudio Scheer[1] and Gabriell Araujo[1]

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

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 divide and conquer architecture. In short, the unsorted vector is divided until it has a specific size, named delta. The execution forms a perfect balanced binary tree. Therefore, the left and right children of a node sort a part of the vector and send it back to the parent. The parent will merge the two vectors received from the children, maintaining the order of the elements, and sent to the parent, until reaching the master node.

Bubble Sort

The bubble sort problem addressed here consists of sorting one vector with 1000000 integers. Figure 1 shows the results of the executions using the sequential (Listing 2) and the MPI version (Listing 3), with different numbers for delta.

Since only the last level of the execution tree will sort the subvectors, the parent levels will not work. This causes an unbalanced exploitation of parallelism. To address this problem, we used a technique to force all the cores to, at some point, sort a subvector. So instead of changing the implementation to force all workers to sort a part of the vector, we simply increase the number of MPI processes (workers). This will force the cores to use hyper-threading or, sometimes, even the time-sharing technique, allowing a balanced exploitation of parallelism.

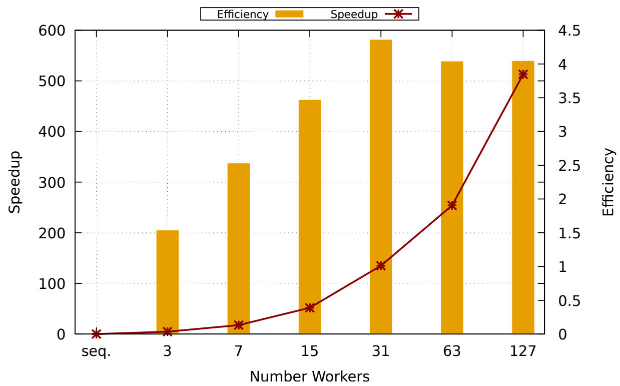

Figure 1: Speedup x Efficiency

When we used 31 workers, each worker had to sort subvectors with 62500 items. Of these workers, at least 7 of them had to be executed using hyper-threading. In addition, we used the other 9 idle cores. These two facts can explain the 83x increase in speedup when using 31 workers instead of 15.

Forcing cores to use time-sharing for some workers has also increased the speedup for the bubble sort algorithm. However, time-sharing reduced the efficiency, as expected, since workers have to wait for preemption to execute their task on the CPU.

The explanation for this high speedup, even when using time-sharing, may come from the nature of the bubble sort algorithm. Bubble sort has a time complexity of $O(n^2)$ for the worst case scenario. Compared to other sorting algorithms, such as quicksort, the time complexity is the same for the worst case scenario. However, bubble sort algorithm is much less complex. This means that smaller subvectors tend to be sorted faster in bubble sort.

Hence, even with the highest message traffic when more workers are used, the subvectors are sorted faster.

Bubble Sort Source Code

Listing 1: Dataset generator

```
#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<int> get_dataset(int vector_size) { return get_vector(vector_size); }
} // namespace dataset
```

Listing 2: Bubble Sort Sequential

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

using namespace std;

vector<int> load_dataset(int vector_size) {
    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    vector<int> vector_unsorted = dataset::get_dataset(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 vector_unsorted;
}

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[d] > v[d + 1]) {
                temp = v[d];
                v[d] = v[d + 1];
                v[d + 1] = temp;
                swapped = 1;
            }
        c++;
    }

    return v;
}

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

    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    vector<int> v_sorted = bubble_sort(vector_unsorted);
    chrono::steady_clock::time_point end = chrono::steady_clock::now();
```

```
    double total_time =
        chrono::duration_cast<chrono::duration<double>>(end - begin).count();

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

Listing 3: Bubble Sort MPI

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

using namespace std;

vector<int> load_dataset(int vector_size) {
    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    vector<int> vector_unsorted = dataset::get_dataset(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 vector_unsorted;
}

vector<int> interleaving(vector<int> vector_left, vector<int> vector_right) {
    vector<int> result;
    std::merge(vector_left.begin(), vector_left.end(), vector_right.begin(),
               vector_right.end(), std::back_inserter(result));
    return result;
}

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[d] > v[d + 1]) {
                temp = v[d];
                v[d] = v[d + 1];
                v[d + 1] = temp;
                swapped = 1;
            }
        c++;
    }

    return v;
}

template <typename T>
std::vector<T> slice(std::vector<T> const &v, int begin, int end) {
    std::vector<T> sliced(v.cbegin() + begin, v.cbegin() + end + 1);
    return sliced;
}

bool is_power_of_2(int x) { return x > 0 && !(x & (x - 1)); }

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

```
    int delta = atoi(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);

    int needed_processes =
        std::pow(2, 1 + std::floor(std::log2(vector_size / delta))) - 1;

    if (!is_power_of_2(vector_size / delta)) {
        cout << "Error: vector size divided by delta must be a power of 2."
             << endl;
        MPI_Abort(MPI_COMM_WORLD, -1);
    } else if (num_processes != needed_processes) {
        cout << "Error: you need " << needed_processes
             << " processes, but only allocated " << num_processes << "."
             << endl;
        MPI_Abort(MPI_COMM_WORLD, -1);
    }

    int parent_node = std::floor(std::abs((my_rank - 1) / 2));
    vector<int> sub_vector;
    vector<int> sub_vector_sorted;

    double begin;

    if (my_rank != 0) {
        MPI_Probe(parent_node, 0, MPI_COMM_WORLD, &status);
        int sub_vector_size;
        MPI_Get_count(&status, MPI_INT, &sub_vector_size);
        sub_vector.resize(sub_vector_size);
        MPI_Recv(&sub_vector[0], sub_vector_size, MPI_INT, parent_node, 0,
                 MPI_COMM_WORLD, &status);
    } else {
        sub_vector = load_dataset(vector_size);
        begin = MPI_Wtime();
    }

    if (sub_vector.size() <= delta) {
        sub_vector_sorted = bubble_sort(sub_vector);
    } else {
        int sub_vector_split_index = sub_vector.size() / 2;
        int left_node = (my_rank * 2) + 1;
        int right_node = (my_rank * 2) + 2;

        MPI_Send(&sub_vector[0], sub_vector_split_index, MPI_INT, left_node, 0,
                 MPI_COMM_WORLD);
        MPI_Send(&sub_vector[sub_vector_split_index], sub_vector_split_index,
                 MPI_INT, right_node, 0, MPI_COMM_WORLD);

        sub_vector_sorted.resize(sub_vector.size());

        vector<int> vector_left(sub_vector_split_index);
        vector<int> vector_right(sub_vector_split_index);
        MPI_Recv(&vector_left[0], sub_vector_split_index, MPI_INT, left_node, 0,
                 MPI_COMM_WORLD, &status);
        MPI_Recv(&vector_right[0], sub_vector_split_index, MPI_INT, right_node,
                 0, MPI_COMM_WORLD, &status);

        sub_vector_sorted = interleaving(vector_left, vector_right);
    }

    if (my_rank != 0) {
        MPI_Send(&sub_vector_sorted[0], sub_vector_sorted.size(), MPI_INT,
                 parent_node, 0, MPI_COMM_WORLD);
    } else {
        double end = MPI_Wtime();
```

```
        double total_time = end - begin;
        cout << "Vector size: " << vector_size << endl;
        cout << "Time sort (s): " << total_time << endl;
    }

    MPI_Finalize();

    return 0;
}
```