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 (2x24 cores) in the Cerrado cluster. 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].

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

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 83.14%. This indicates that, up to 12 processes, the bubble sort algorithm can exploit up to 83.14% of the expected speedup.

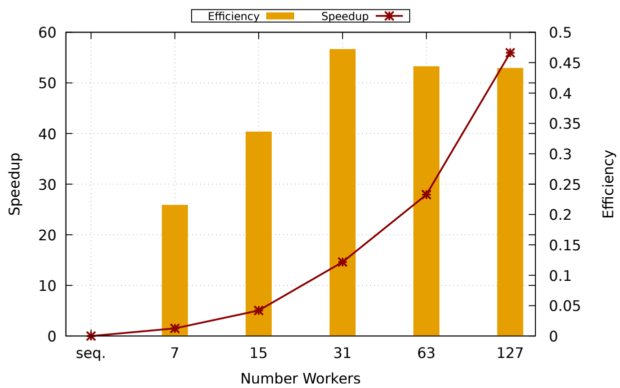

Figure 1: Speedup x Efficiency

Figure 1 shows that the speedup grows linearly for physical cores. However, when using hyper-threading, the speedup is only 28.31% 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.

Discussion

In linear regression, hyper-threading increased the execution time when compared to the approach using all physical cores and the efficiency was only 3.9%. For bubble sort, hyper-threading has almost the same speedup as using all physical cores, but not as efficient as using only physical cores. Therefore, in general, hyper-threading can have a speedup, but efficiency tends to decrease. 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 and begins to decrease.

[1]What a horrible scenario!

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]);
```

```cpp
    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;
}
```