INFO-F-404 – REAL-TIME OPERATING SYSTEMS

PARALLEL COMPUTING: K-NEAREST NEIGHBORS

EL MIRI Hamza
000479603
KEZAI Wassim
000357375

17/12/2021

Contents

1 Introduction:

The goal of this project was to implement a solver for the *k-nearest neighbors* problem and to speed up the computations by making use of the parallelization capabilities of the *MPI* library.

The working language is C++ and we will go over the important implementation decisions as well as solutions to the problems encountered.

2 Implementation

2.1 Overview

We chose to parallelize the computation of the distances: Each node, including the root node, will receive a subset of the points listed in the environment file. Afterwards, each point to be classified is broadcasted to all nodes.

Each node computes the distance between the point and all others in the subset of the environment it has received and finally, the results are gathered in the root node where they'll be sorted and the point will be classified according to the k first elements in the sorted array.

The MPI functions used were:

- `MPI_Scatterv` to evenly distribute the points across all nodes and to handle the case where the number of points isn't divisible by p which `MPI_Scatter` can't do.
- `MPI_Send` and `MPI_Recv` are used before the scatter operation to communicate to each node the number of elements that it will receive so that a large enough buffer can be allocated because some nodes will receive at most 1 more element than the others when n isn't divisible by p.
- `MPI_Gatterv` to gather the results of the computations by each node into the root node.
- `MPI_Bcast` to efficiently communicate the size of a point to allocate the destination buffer, as well as its coordinates.

2.2 Workload balance

Let n be the number of points in the environment, p the number of nodes and m the number of points to be classified.

By distributing the workload in this way, we reduce the time complexity of the parallel algorithm compared to the sequential algorithm by a factor $\frac{1}{p}$. Assuming the sort is done in $\mathcal{O}(n\log(n))$:

$$\mathcal{O}(m \times n \times n\log(n)) \rightarrow \mathcal{O}(m \times \frac{n}{p} \times n\log(n))$$

2.3 Speedup factor

Excluding the initial setup, the fraction f of sequential computations is approximately 6% (Gather + Sort + Write result). Therefore, the speed up factor is bound by:

$$\lim_{n\rightarrow+\infty} S(n) = \frac{1}{f} = \frac{1}{0.06} = 16,66..$$

3 Additional work

3.1 Generator

A random environment and challenge set generator was written for testing.
Usage:

- Generate a random environment:

```
./generator [colors] [n_points] [dim] [output]
```

- Generate a random challenge set:

```
./generator [n_points] [dim] [output]
```

  Where

  - colors: Integer specifying the upper bound of the color range
  - n points: Total number of points in the environment

- dim: Dimensions of a point. Each coordinate will be a random float $\in$ [-100, 100];
- output: Destination file

4 Difficulties and potential improvements

After finishing the computations, there was no way of informing the other nodes to terminate their execution resulting in a deadlock. This problem was handled by broadcasting in advance the number of points to be classified allowing all nodes to terminate in a synchronous way after receiving the last point.

The algorithm can be slightly improved by parallelizing the sequential sort in the root node:

Let n m, and p be the number of points in the environment, the number of points to be classified and the number of nodes respectively. By allowing each node to sort its own results, we can reduce the time complexity of the sort from $\mathcal{O}(n\log(n))$ to $\mathcal{O}(\frac{n}{p}\log(\frac{n}{p}))$. The sorted results can then be merged by the root process in linear time bringing down the time complexity to:

$$\mathcal{O}(m \times \frac{n^2}{p^2}\log(\frac{n}{p}))$$

5 Conclusion

This project was a good opportunity to put parallel computing techniques into practice, as well as a good introduction to the MPI library, which we used to speed up a computationally costly algorithm.