MPI brute-force all-kNN search

Ioannis Stefanidis - 9587

Aristotel University of Thessaloniki

8 January 2023

For this assignment we were asked implement in MPI a distributed brute-force all-KNN search algorithm for the k nearest neighbors (kNN) of each point in a dataset.

Communication in a Ring

Each MPI process P_i is reading a part of the provided dataset and it calculates the distance of its own points from all (other) points and record the distances and global indices of the k nearest for each of its own points. While calculating the distances, each process sends it's own points to the next process and receives the points from the previous process. By doing that asynchronous we hide the communication costs for these data transfers.

Having NP number of processes means that after NP iterations of sending and receiving all processes would have checked their points against the whole dataset. At this points the host machine (process #0) writes it's own results to a file and one by one receives and appends the results from all other processes.

kNN Search Algorithm

Finding the distances

To find the k nearest neighbors for each point we calculate the Euclidean Distance Matrix D, then sort each row (in parallel with opencilk) using quickselect to get the k smallest distances and finally sort each row from 0 to k using quicksort. To calculate the matrix D the openblas library is used to multiply the matrix X with Y, where X are the process's points and Y the points that are cycling through the ring. After that for every point $d_{ij} \in D$ the addition $d_{ij} = d_{ij} + \sum_{0}^{d} x_{id}^2 + y_{jd}^2$ is performed. So the final matrix D can be expressed as (the -2 multiplication is performed by `openblas`):

$$D = (X \circ X)\,1_{d \times n} - 2XY^T + (Y \circ Y)\,1_{d \times m}$$

Keeping track of global indices

When the matrix D is created we also create a matrix D_{ind} to keep track of the indices. While sorting the D with quickselect and quicksort the same swaps are preformed on the *D_ind* matrix, this way we can keep track of the global indices.

Testing

To check the correctness of the program regular Cartesian 2D,3D, and 4D grids were used as inputs. For these grids the first 3^d neighbors are known for each point, so the program output for $k = 3^d$ was compered with the known neighbors and found equal.

Benchmarking

To measure the performance and speedup we used the following datasets:

- MNIST $60K \times 784$
- 2D grid $1M \times 2$
- 3D grid $1M \times 3$
- 4D grid $810K \times 4$

These grids were generated using the `grid` executable[1] which is provided with the source code.

Finding the fastest config

The AUTH HPC[2] provided us maximum of 1024 cpus. Each node in the *rome* partition has 128 cpus so to fully take advantage of the Aristotel cluster we need to use 8 nodes. So we ran 4 tests for each dataset, in each test we assigned a different amount of cpus to every process. The fastest results of these tests are shown at the following table (Fig. 1), where *ppn* is the number of process per node and *cpp* is the number of cores per process ($ppn = 128/cpp$).

Dataset	m	d	k	*cpp*	*ppn*	time(s)
2D-Grid	1M	2	9	4	32	402
3D-Grid	1M	3	27	4	32	117
4D-Grid	810K	4	81	2	64	62
12D-Grid	531K	12	10	2	64	13
MNIST	60K	784	3	8	16	5.65

Figure 1: Table of the fastest configs for each dataset

[1] e.g. `./grid 3 100 100 100` outputs the 3D grid.

[2] HPC website: https://hpc.auth.gr/

By visualizing the results (Fig. 2), we see that creating less processes per node with greater amount of cpus is slower than more processes with smaller amount of cpus. Also we notice that for the *MNIST* dataset that has much less points than the others but bigger dimension, the fastest config uses 8 cores per process, that could be because the openblas library the multiplies $X(m \times d)$ with $Y(n \times d)$ needs the extra cores.

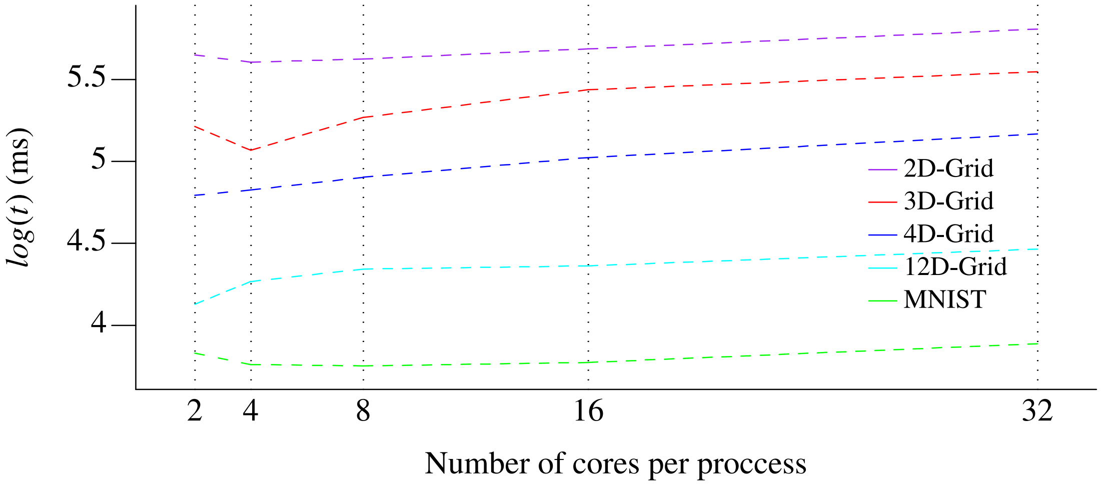

Figure 2: Logarithmic execution time for different ammount of cores per proccess

Finding the most efficient config

Utilizing all 1024 cpus available give us the best results but lets try to find the most efficient config by looking at the strong scaling. In strong scaling, the problem size is kept constant while the number of nodes is increased.

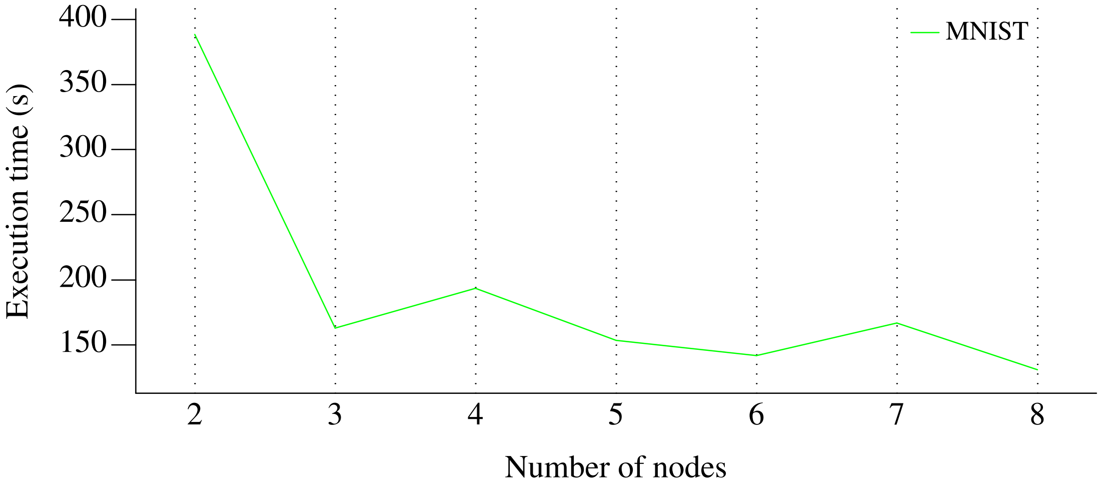

Figure 3: Execution time for the MNIST dataset with respect to the number of nodes (cpp=8, ppn=1)

From the above graph we notice that after 5 nodes the decrease in execution time levels off, so we can maybe choose 6 nodes as the saturation point for the MNIST dataset, where adding more nodes doesn't result in significant decrease in execution time.

Because of the unavailability of resources in HPC, finding the saturation point for each dataset wasn't possible. In the table below (Fig. 4) you can find all the tests that took place for this assignment (everything ran in the rome partition of HPC that has *AMD EPYC 7662* cpus).

Dataset	m	d	k	Nodes	*cpp*	*ppn*	time(s)
2D-Grid	1M	2	9	8	32	4	641.4
2D-Grid	1M	2	9	8	16	8	485.8
2D-Grid	1M	2	9	8	8	16	420.4
2D-Grid	1M	2	9	8	4	32	402.6
2D-Grid	1M	2	9	8	2	64	446.1
3D-Grid	1M	3	27	8	32	4	352.0
3D-Grid	1M	3	27	8	16	8	273.7
3D-Grid	1M	3	27	8	8	16	185.4
3D-Grid	1M	3	27	8	4	32	117.0
3D-Grid	1M	3	27	8	2	64	163.4
4D-Grid	810K	4	81	8	32	4	146.9
4D-Grid	810K	4	81	8	16	8	105.4
4D-Grid	810K	4	81	8	8	16	79.9
4D-Grid	810K	4	81	8	4	32	66.9
4D-Grid	810K	4	81	8	2	64	62.0
12D-Grid	531K	12	10	8	32	4	29.1
12D-Grid	531K	12	10	8	16	8	23.0
12D-Grid	531K	12	10	8	8	16	22.0
12D-Grid	531K	12	10	8	4	32	18.5
12D-Grid	531K	12	10	8	2	64	13.4
MNIST	60K	784	3	8	32	4	7.7
MNIST	60K	784	3	8	16	8	5.9
MNIST	60K	784	3	8	8	16	5.6
MNIST	60K	784	3	8	4	32	5.7
MNIST	60K	784	3	8	2	64	6.7
MNIST	60K	784	3	2	8	1	388.4
MNIST	60K	784	3	3	8	1	162.9
MNIST	60K	784	3	4	8	1	193.4
MNIST	60K	784	3	5	8	1	153.5
MNIST	60K	784	3	6	8	1	141.9
MNIST	60K	784	3	7	8	1	166.8
MNIST	60K	784	3	8	8	1	130.9

Figure 4: All the runs that were recorded during this assignment.

The source code for this assignment is available at:
https://github.com/johnstef99/mpi-nextdoor