Parallel and Distributed Computing
Spring Semester - 2020/2021

k-Nearest Neighbors: The Ball Algorithm

Group 15 - Diogo Moura (86976), Francisco Romão (90074), João Figueiredo (90108)

22nd May 2021

I. Serial Implementation

The serial implementation remains almost unchanged from the one delivered in the first part of the project. Only differs in the $recursive\_furthest\_apart()$ algorithm. It was altered from recursive to simply considering the two furthest apart points the one furthest from the one in vectors first position and the more distant to this.

II. OpenMPI Implementation

A. Approach

The pure OpenMPI implementation alters the previously implemented algorithm by running on MPI in multiple processes simultaneously. It separates the nodes by creating two communicators, one for each child, every time the tree splits. This way, when the level of the tree has a number of nodes equal to the number of MPI processes, each process computes a distinct part of the tree and doesn't communicate anymore with the other processes. When there are multiple processes on a tree level, they all perform the same computations to avoid the communication overhead of broadcasting the points at the end .Each time the communicators are split, the ranks of each processor are changed.

We also tried another version where the nodes were computed only by the $rank = 0$ process. The processes that had rank different from 0 helped the $rank = 0$ process on the furthest_apart algorithm, where an approximation for the furthest_apart points is computed. In this process, the $rank = 0$ processor broadcasts a random point to all the other processors and all processors compute the maximum at a portion of the total points, At the end, a reduction is performed to get the global maximum point and its index. However, this version was slower than the previous since there was a high communication to broadcast the results of the points. The communication cost and the cost of keeping some processors stalled during the execution of the other tasks was higher than the cost reduction of performing this function in a distributed fashion.

B. Load Balancing

The described load balancing approach depends on the tree balancing, shaped by the number of points in the problem. In the beginning of the execution, the computation is the same and redundant for all MPI processes. All processes have the same workload, so that the $rank = 0$ process doesn't have to broadcast ordered points to the other processes, this is in order to avoid the communication overhead.

By the time the number of processes equals the number of nodes in the tree level, all processes work in different nodes independently. Each node computation load at each tree level is almost exactly the same due to the very nature of the algorithms executed, whose load is independent of the values contained by the poins.

Each subtree also takes almost exactly the same load, since the splitting of the points by its median balances exactly each subtree when the number of points is even and produces an unbalance of one point on the load when the number of points is odd.

The optimal balance is only achieved for number of points that are powers of 2. Consequently, for almost all the number of points, a perfect load balancing isn't achieved, However, a very good load balancing is achieved, some of the processes might get just very few more tree levels and additional very few node computations compared to the total number of node computations.

C. Synchronization

An explicit barrier was also used for run-time measurement purposes, safeguarding that all the processes

finish before assuming the the tree is built, and the runtime measurement is finished.

Synchronization is required just on just one cases for our implementation.

Since each MPI processes computes all nodes, they don't rely on others computations to move onto the next tree node, except to produce the communication split. In fact, this is all the synchronization performed at our implementation.

To save on communication overhead, the tree nodes are printed on the MPI process that computed them, so this step doesn't require synchronization.

D. Limitations and alternative experiments

The described implementation suffers several limitations. Namely:

- Inability of solving problems that do not fit in the memory of a single machine;
- Wasted computational power computing redundant tree nodes in the beginning of the problem;

To address the first limitation, the $gen\_points.c$ was altered so that each MPI process generates only a portion of the points ($\frac{n_{points}}{n_{procs}}$). The implementation of this process is found in the file $distributed\_mpi/algoritmosNaoIntegrado/genpoints.c$.

To address the second limitation, a Parallel Sorting with Regular Sorting algorithm was developed. This is described on section IV. Its implementation can be found in the file $distributed\_mpi/algoritmosNaoIntegrados/psrs\_mpi.c$. The conjugation of the PRSR algorithm and the final ball algorithm was not done because of the short time.

The fact that our implementation achieves good load balance without little to no synchronization and communication overhead is the reason why very good speedups are obtained. A significant speedup improvement would be difficult to obtain if all the processes mantain a local copy of all the points.

However, by distributing the points by each computer at the start of the tree construction and performing all the operations in distributed fashion, such as described in the two addressed solutions, would yield significantly better results.

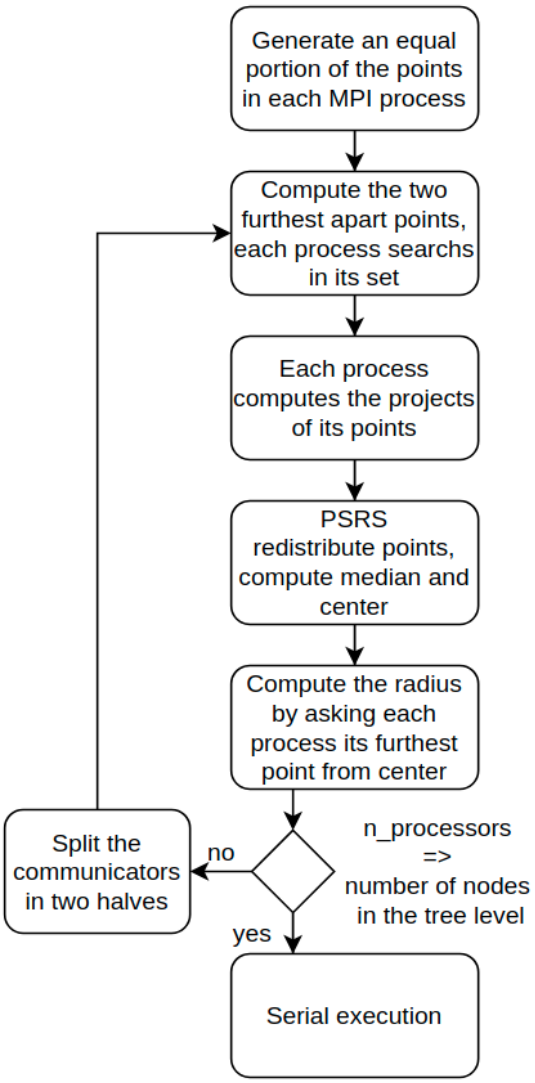

Fig. 1. Alternative implementation tried

III. Hybrid OpenMPI and OpenMP Implementation

The implementation that combines OpenMPI and OpenMP used the OpenMPI approach previously mentioned in the section II. The parallelization, is implemented both on the construction of tree (using task), in the functions used to compute the furthest point from another and in the function to compute the pseudo projections (returns values that allow to order the projections) in the line.

This doens't pose any problems in load balancing or synchronization.

When the MPI communication splitting is over and each processor each half of the available threads are assigned to each child of the current tree node until each child as only one thread.

The OpenMP parallelization, at function level, is performed using simple *parallel for* constructs, with reduction performed at the end for the furthest point calculation and with no reduction for the projections calculation. These two parallelizations present almost no synchronization and also achieve a good load balance. To note that, when the MPI communication split ins't over the parallelization at the function level is always performed. However, after each processor is working independently this only occurs for the first few levels of the tree construction, after which the parallelization is performed only at the tree level as described in the previous paragraph.

The OpenMP parallelization approach used in our implementation is good from the point-of-view of synchronization and load balancing due to the tree splits being almost perfectly balanced, due to the same reasons described in section II-B.

IV. PSRS Algorithm

Parallel Sorting Regular Sampling is an algorithm used to sort a vector of values distributed over all machines.

Firstly, with the values already distributed, each machine takes a sample of equidistant points. The method for obtaining the samples and afterwards the partition limits could be sorting the vector and taking the middle elements, however our implementation uses the $QuickSelect$ algorithm. This method avoids sorting the algorithm which saves execution time.The size of the sample can be varied to obtain the better results. The samples are gathered on root machine - in our case the root machine is the one that has the rank 0. Then the root node takes the values that will limit the partitions on the nodes. These values have to be has equidistant has possible, making the end values of each node close, in number. After the nodes receive the limits, using a simple Broadcast, their vectors are individually sorted and sent to the respective machine i.e. the first partition is sent to the node with the rank 0, the second to rank 1 and so on. To implement this part each machine was allocated a new vector that will hold the conjugated partitions received, the size of this array was found by testing, since it is not guaranteed that at the end of the algorithm every machine have the same number of points, with higher probability that they won't. At this point, with the new partitions, it is faster to find the median value and the separation of the nodes in two children is almost automatic since the nodes will only have to be divided in lower and higher halves. With the exception of the center node that will have to divide its value since the most probable case it to the median not to be on the edge of a vector of points of one machine.

The sample size was determined by testing it is concluded that a sample of size equal to the number of nodes+1 is a good balance between computational time and a reasonable final distribution of values between nodes. The root node always has a smaller number of points, but this could be because the sample are taken starting from zero, which mean that the last partition almost always end up with a more points than the other partitions.

V. Analysis of the results

On table I the results for the implementation that uses just MPI are shown.

On table II the results for the hybrid implementation that uses MPI+OpenMP are shown.

Better speedups are achieved with the hybrid approach, as expected.

The speedup is far from being linear with the number of processors used, due to the fact that the nodes that are more difficult to compute are the first ones, where the computations are replicated and redundant between processors.

VI. Conclusion

The fact that our implementation achieves good load balance without little to no synchronization and communication overhead is the reason why very good speedups are obtained.

Better results could be obtained, using the functions described in *Limitations and alternative experiments* section of the report.

However, due to deadline constraints and due to problems in integrating the developed solutions into the code of our previous serial implementation, we preferred not to do it. Namely the way the index of the points were used at almost all of the functions of the serial version, would imply deep changes in almost all of our code, since when the points are distributed and non-replicated this approach can't be used anymore.

VII. Performance Results

TABLE I

RUN-TIMES AND SPEEDUPS FOR THE OPEN**MPI** VERSION (EACH TASK HAS 4 THREADS ASSIGNED)

Distributed - OpenMPI							
Arguments	T_{serial}	T_{D1proc}	$S_{D1procs}$	$T_{D2procs}$	$S_{D2procs}$	$T_{D4procs}$	$S_{D4procs}$
20 1000000 0	5.3	4.8	1.10	3	1.77	2	2.65
50 1000000 0	9.8	8.1	1.21	5.4	1.81	3.7	2.65
3 5000000 0	13	12.9	1.01	6.9	1.88	4.3	3.02
4 10000000 0	31.2	38.7	0.81	17.1	1.82	10.1	3.09
3 20000000 0	63.2	44.9	1.41	26.5	2.38	20.1	3.14
4 20000000 0	69	52.7	1.31	29	2.38	21.9	3.15
$T_{D8procs}$	$S_{D8procs}$	$T_{D16procs}$	$S_{D16procs}$	$T_{D32procs}$	$S_{D32procs}$	$T_{D64procs}$	$S_{D64procs}$
1.5	3.53	1.2	4.42	0.9	5.89	0.9	5.89
2.8	3.50	1.9	5.16	1.8	5.44	1.9	5.16
2.9	4.48	2	6.50	1.9	6.84	2.1	6.19
5.3	5.89	4	7.80	3.4	9.18	5.5	5.67
10.3	6.14	9.4	6.72	7.9	8.00	7.8	8.10
14.1	4.89	8	8.63	8.7	7.93	8.7	7.93

TABLE II

RUN-TIMES AND SPEEDUPS FOR THE HYBRID VERSION (EACH TASK HAS 4 THREADS ASSIGNED)

Hybrid - OpenMPI and OpenMP							
Arguments	T_{serial}	T_{D1proc}	$S_{D1procs}$	$T_{D2procs}$	$S_{D2procs}$	$T_{D4procs}$	$S_{D4procs}$
20 1000000 0	5.3	1.8	2.94	1.2	4.42	1.1	4.82
50 1000000 0	9.8	3.2	3.06	2.2	4.45	1.5	6.53
3 5000000 0	13	4	3.25	2.5	5.20	1.7	7.65
4 10000000 0	31.2	8.7	3.59	5.4	5.78	3.7	8.43
3 20000000 0	63.2	17.5	3.61	10.7	5.91	7.4	8.54
4 20000000 0	69	18.2	3.79	11.2	6.16	7.6	9.08
$T_{D8procs}$	$S_{D8procs}$	$T_{D16procs}$	$S_{D16procs}$	$T_{D32procs}$	$S_{D32procs}$	$T_{D64procs}$	$S_{D64procs}$
0.7	7.57	0.7	7.57	0.7	7.57	0.8	6.63
1.3	7.54	1.7	5.76	1.3	7.54	1.6	6.13
1.4	9.29	1.8	7.22	1.3	10.00	1.8	7.22
2.9	10.76	3.6	8.67	2.8	11.14	3.5	8.91
5.8	10.90	6.7	9.43	5.5	11.49	6.8	9.29
7.2	9.58	5.5	12.55	5.6	12.32	6.8	10.15