Parallel & Distributed Systems

First Nearest Neighbor with MPI

Marios Mitalidis
Aristotle University of Thessaloniki
mariosam@ece.auth.gr

04/11/2016

Abstract

The implemented algorithm concerns a set of points C in the 3-dimensional space, and more specifically, finding the nearest neighbour for each point in a different set of points Q. In the simple case, the complexity is O(|C|.|Q|), since for each point in Q, we access each point in C to find the nearest neighbour. We can impove the execution time if we split the search space in boxes, so that each box contains a subset of C. Thus, we now search in the neighbourhood of each point in Q.

Algorithm

The algorithm first generates the sets C and Q, so that the point coordinates follow a uniform districution in [0,1). The points are ditributed among P processes. Then, it splits the search space with a grid of boxes. The boxes are assigned to the processes so that each process has the same number of boxes. In this stage the points are re-distributed as well. Finally, for each point q in Q the process computes its neighbouring boxes and after communicating with other processes computes the nearest neighbour.

Parallelism Method

In order to take advantage of a number of computing units, the processing is assigned to different processes which cooperate with each other with the MPI protocol. In practice, each process stores in its local memory a different subset of C, which corresponds to specific boxes, and a subset Q.

Examining the problem from the point of view of process A, we notice that it should do the following:

1. For each point q, that process A tries to solve the problem, make a list of the processes that A should communicate with. Those are the processes that are assigned to boxes in the neighbourhood of q.
2. Prepare a message for each process.
3. Communicate with each process.
4. Collect all the answers and compute the minimum distance.

The messages have the following form:

- *Process A to B:* For each of the points I will send you, reply with:
    1. 1. the point with the minimum distance

    2. the minimum distance
    from all of your points.

- *Process B to A:* Answering to request from process A.
- *Change of roles:* Process A responds to process B request.

Moreover, given that each process needs to communicate with each other process, a synchronisation method is necessary. First, we notice that each process has a unique id in the interval *[0, nproc-1]*. Our synchronisation method is based on spliting message passing between processes in rounds. During each round a process either has one partner to communicate with or proceeds to the next round. Given the process with id = i, being in round = incr, then it should communicate with the process with id = incr-i. Thus, we uniquely define the partner of each process and form each possible pair. A part of the implementation is presented in the following box:

```
void Result_par::comm_and_solve(const World& C) {

    for (int incr = (numtasks-1) + (numtasks-2); incr >= 0; --incr) {

            int partner = get_partner(incr);
            if ( partner != -1) {
                    if (rank > partner) {
                            send_msg_to_proc(C, partner);
                            recv_msg_from_proc(C, partner);
                    }
                    else {
                            recv_msg_from_proc(C, partner);
                            send_msg_to_proc(C, partner);
                    }
            }
    }
}
```

Communication sychronisation between processes.

Correctness check

In order to ensure the functional correctness of the code, various tests were performed during code development. The purpose of the tests was to confirm assumptions about the code functionality.

More specifically, we confirmed that a process can find the nearest neighbour within its own set of points. In other words, a serial implementation was developed.

As far as the message passing is concered, we confirmed that the processes communicate as expected, by printing proper messages.

Finally, the results of both the serial and parallel implementation were compared, for a range of values, with a simple serial reference implementation, with O(n^2) complexity, that was developed in Octave. This concludes the correctness check.

Results

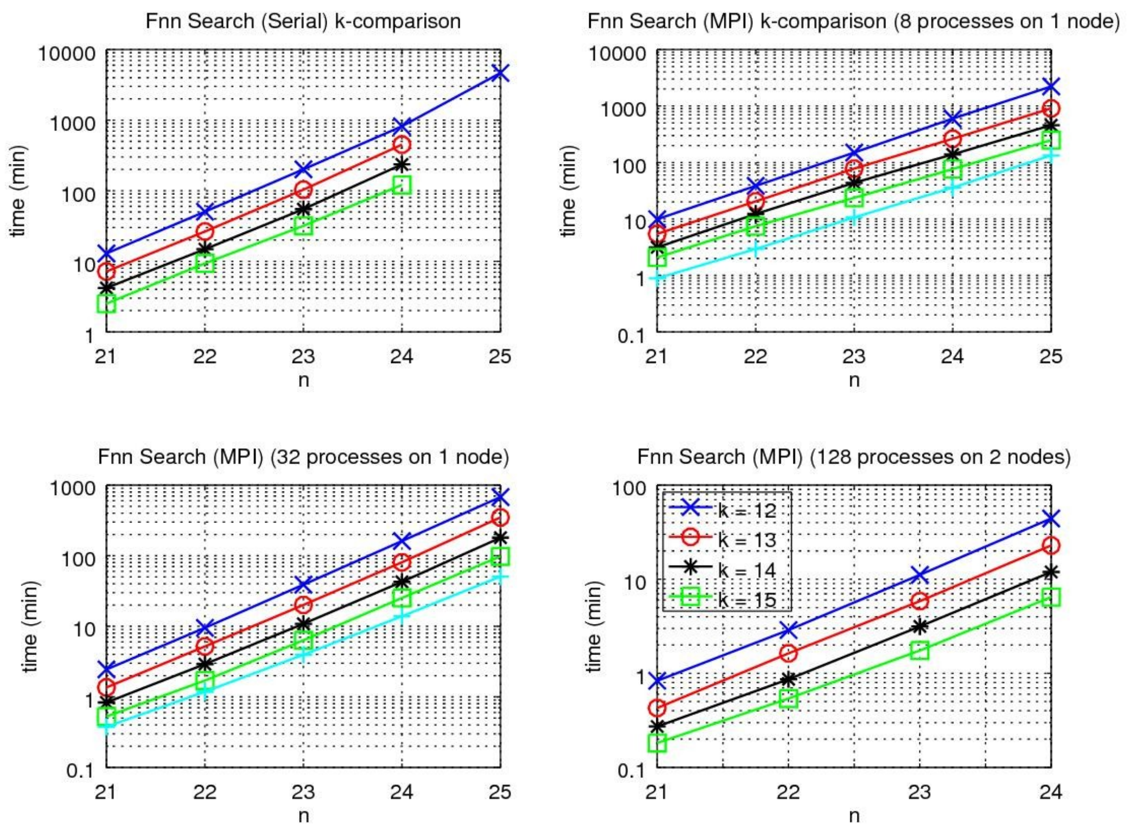
For each run of the algorithm, we define the number of points in set C to be |C| = |Q| = 2^n , the number of boxes to be K = 2^k and the number of processes P = 2^p.

The experiments were carried out at the HPC of AUTh. The worker nodes of the HPC have either 64 or 8 cores, and the cpu model is AMD Opteron(TM) Processor 6274, with x86_64 architecture.

We run approximately 50 experiments for many combinations of (n,k) and different p values. Moreover, we examine the relationship of the parameters "number of nodes per job (nodes)" and "processes per node (ppn)". For the measurement of the execution time, we do not include the time it takes to create C and Q.

Effect of k parameter

The first result concerns the effect of k parameter, that is the number of boxes that split the point in C. From the following diagram, we notice that independently of the number of processes and for each value of n; execution time is reduced, as we increase k.

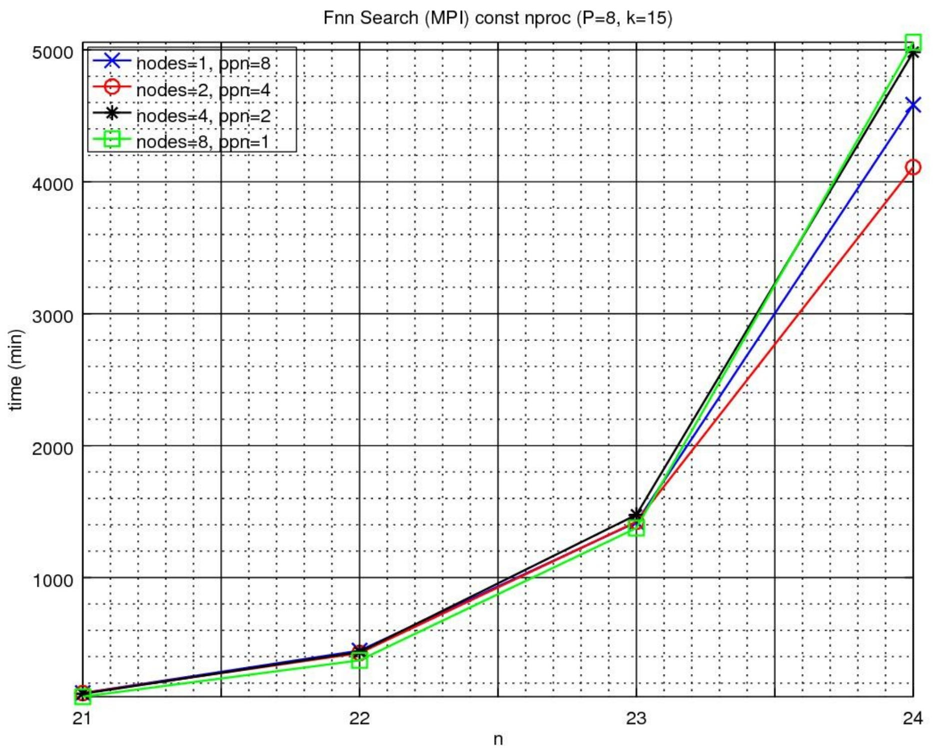


Effect of k parameter

We anticipated this result, since the number of points included in the neighbourhood of a point q is reduced every time we split the search space with more boxes.

Parameters *nodes* and *ppn*

The second experiment is about program efficiency for a given number of processes that run either locally, or on different worker nodes. Thus, for P = 8 and for the pairs (nodes, ppn) = (1,8) , (2,4) , (4,2) , (8,1), we have the following execution times.



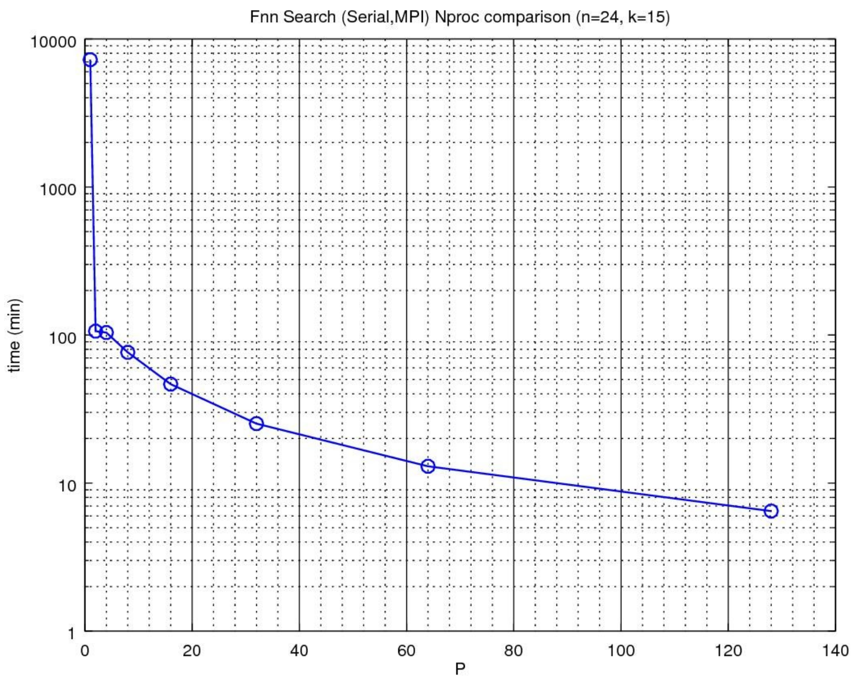
Execution time for different (nodes,ppn) pairs

As we can see from the diagram, the case (8,1) (green curve) is approximately as fast as the other cases for n = 21, but becomes slower for n = 24.

This could be explained in the following way: first of all we take into consideration that the processes which are executed on different nodes are communicating slower, since they are linked with the Gigabit Ethernet network. Thus, when we use 4 or 8 nodes and need to exchange large volumes of data, the total time increases.

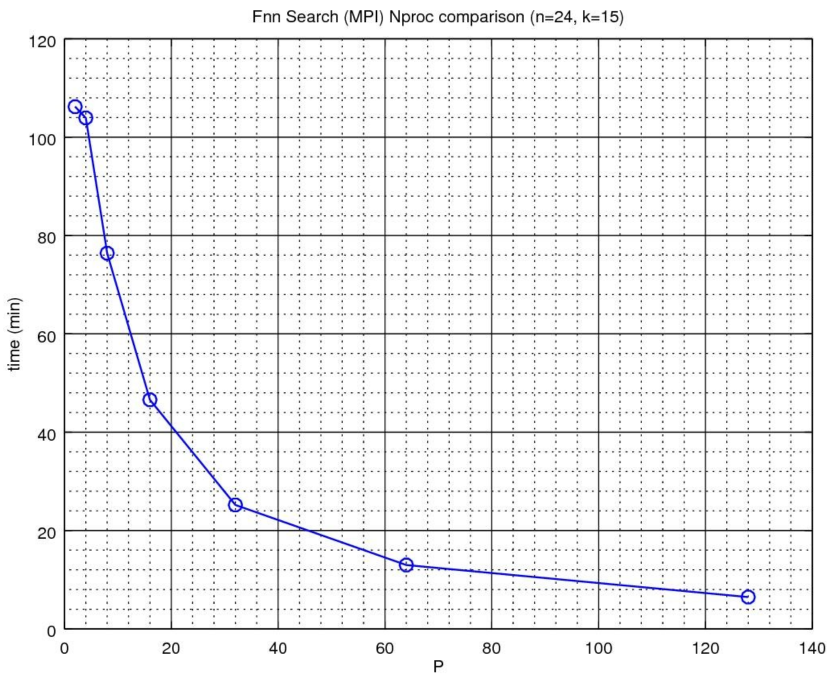
Total elapsed time

Finally, for the extreme case of (n=24, k=15) we notice a consistent decrease in execution time as we increase the number of processes. For the most effective use of the cluster nodes we choose the pairs (nodes,ppn) to be: (1,1) , (1,2) , (1,4) , (1,8) , (1,16) , (1,32) , (1,64) , (2,64).



Execution time (Serial and MPI) for different P (number of processes)

From the above diagram we notice a decrese in time by two orders of magnitude between the serial and parallel implementation for P = 2. Finally, from the diagram of the parallel implementation we notice that the execution time continues to decrease even for large values of P.



Execution time (MPI) for different P (number of processes)

In the following tables we present the execution times (sec):

k = 12

n/P	1	2	4	8	16	32	64	128
21	775.413	746.064	739.65	559.572	322.69	146.966	91.6293	50.0058
22	3029	2869.32	2777.48	2200.02	1262.92	570.589	348.682	173.052
23	12048.7	11392	11075.7	8738.06	5087.16	2331.86	1349.38	668.02
24	49113.5	45811.4	43955.2	32159	20671.7	9674.11	5381.06	2655.72
25	279328	187808	182909	103472	73117	40652.5	20213.3	

k = 13

n/P	1	2	4	8	16	32	64	128
21	430.022	393.569	376.179	325.755	183.913	81.6278	50.5825	25.6916
22	1585.65	1559	1475.98	1167.84	677.179	311.323	182.751	97.881
23	6209.39	5939.53	5659.88	4591.92	2592.69	1203.57	709.283	350.859
24	26977	23663.2	22467.8	13920.1	10135.3	4842.92	2737.23	1371.92
25		93785.1	92729.7	52472.1	39738.5	20927.8	11365.5	

k = 14

n/P	1	2	4	8	16	32	64	128
21	251.673	233.907	213.25	187.173	106.849	49.9784	30.5129	16.3452
22	889.553	828.572	791.438	686.547	381.243	174.537	102.325	51.9202
23	3286.6	3133.52	3027.08	2485.84	1440.78	650.704	381.713	188.928
24	14157.6	13705.6	11837.9	7593.29	5215.54	2555.86	1438.24	716.541
25		49483.8	48113.2	26861.9	17360.5	10791.1	5093.35	

k = 15

n/P	1	2	4	8	16	32	64	128
21	150.965	143.589	132.414	123.986	66.9374	31.6438	19.7554	10.908
22	560.889	476.897	437.818	427.284	227.244	102.225	60.891	32.4048
23	1905.85	1736.55	1613.46	1422.89	803.443	379.536	225.791	104.949
24	7246.44	7882.04	8427.87	4110.69	2795.01	1511.92	778.792	387.937
25				14595.4	9322.53	5873.19	2675.23	

k = 16

n/P	1	2	4	8	16	32	64	128
21				49.8291	33.5838	22.5986	11.3531	
22				161.341	108.178	70.8804	35.4217	
23				544.51	380.56	236.842	107.907	
24				1984.49	1368.81	831.499	382.255	
25				7635.42	5041.34	3035.94	1435.12	

Conclusion

A parallel implementation of the First Nearest Neighbor algorithm with MPI was studied. The search space was split with boxes, so that we don’t search the entire domain for the nearest neighbor. The experiments from the AUTh HPC indicate a significant reduction in execution time as the number of processes increases.