Project Checkpoint(Nov.18th)

Updated Schedule

So far, we have kept pace with the planned schedule. Specifically, we now have a working version of sequential version, a G-DBSCAN and the python `sklearn` package implementation as reference

Week	Goal	Detail	Progress
Week 1(10/29-11/05)	Research	Write proposal, read related paper and implement sequential version.	Done
Week 2(11/05-11/12)	1st Parallel Implementation	Implement G-DBSCAN with CUDA and do analysis.	Done
Week 3(11/12-11/19)	Checkpoint!	Check point report	Done
Week 4(11/19-11/26)	G-DBSCAN speed & MPI version draft	1. N-Body data-parallel approach to neighbor construction (Sailun) 2. K-D tree approach to neighbor construction(Yueni) 3. Drafting MPI approach to RP-DBSCAN(both)	TODO
Week 5(11/26-12/03)	MPI version finalize & Performance analysis	1. Implement full RP-DBSCAN with MPI and all relevant tricks(Sailun) 2. Fine tune each algorithm and optimize the relevant hyperparameters(Yuenil) 3. Design more test cases, and anlyze the effect of `epsilon` and `minPts` on different algorithms(Yueni)	TODO
Week 6(12/03-12/10)	Final Report Poster	1. Run a grid of experiments setting configurations, and draw graphs for comparison (both) 2. Final report (both) 3. Poster(both)	TODO

As mentioned before, we have so far kept pace with the planned schedule, but largely due to that all algorithms we have implemented so far are on the relatively easy side, and the true challenge will be RP-DBSCAN and we expect to spend roughly two weeks on that. Also, we expect to improve the G-DBSCAN by using either the k-d tree or the N-Body data-parallel approach mentioned in the lecture.

For the poster session: we plan to explain our approach to the problem, and the optimizations/tricks we have used throughout the implementation. We will also show graphs which compare the various aspects (speedup, time breakdown, load balance, scalability, and memory footprint) of different scan algorithms on different scenario.

Preliminary Result and Issues

We start from scratch and build a workflow of:

- implementing a new scan algorithm class that inherits the pure virtual base class `DBScanner`, so far we have built:
    - `SequentialDBScanner`: a naive implementation where we build `neighbors` by going through every pair of points (a complexity of `O(n^2)`), then we simply find all connected parts by performing BFS
    - `Seq2DBScanner`: a sequential version of the algorithm mentioned in G-DBSCAN, notably, it performs worse than the `SequentialDBScanner` for seeking the possibility of parallelization.
    - `ParallelDBScanner`: a cuda version of the G-DBSCAN, which utilizes a compact adjacency list to represente the graph. Both graph construction (exlusive scan with Thrust library) and cluster identification (BFS with level synchronization) is parallized.
    - `RefScanner`: basically we invoke the `sklearn.cluster.DBSCAN`. For reference, this version includes a k-d tree for building `neighbors` faster (`O(n^2)` on average, and `O(n*log n)` empirically). Also, the BFS procedure is optimized using `Cython` (`c` extension for `python`).

- including a new test case with different number of points and scatter pattern, so far we have the following test cases:
    - `random-{k}` where `k` in `{1e3, 1e4, 1e5, 1e6}`, we sample uniformly randomly from the `([-10,10], [-10, 10])`. We are expecting to build more test cases such as `ring`, `mixture`, but so far we only use the random case for testing the correctness by checking the output labels of our scanner against the `RefScanner`.

The following is a summary of the runtime (in `ms`) of each of the combination of (`scannerType`, `testCase`):

	RefScanner	ParallelDBScanner	SequentialScanner	Seq2DBScanner
random-1000	12.661934	1020.34	**4.78667**	12.2835
random-10000	**115.019083**	139.35	206.94000	454.4650
random-100000.in	3408.552885	**468.22**	13815.60000	40600.5000

Due to the small overhead, `SequentialScanner` performs the best for 1000 points, but `RefScanner` and `ParallelDBScanner` performs better on larger problems. Yet we have not include either the k-d tree/N-body data-parallel approach to speed up the `neighbors` construction yet, we expect that to bring

even more advantage for `ParallelDBScanner` and set a strong baseline.
Also we notice a dubious large runtime for `ParallelDBScanner` on `random-1000`, we will further look into it this week.

References

[1] Ester, M., Kriegel, H.P., Sander, J. and Xu, X., 1996, August. A density-based algorithm for discovering clusters in large spatial databases with noise. In Kdd (Vol. 96, No. 34, pp. 226-231).

[2] Andrade, G., Ramos, G., Madeira, D., Sachetto, R., Ferreira, R. and Rocha, L., 2013. G-dbscan: A gpu accelerated algorithm for density-based clustering. Procedia Computer Science, 18, pp.369-378.

[3] Patwary, M.A., Palsetia, D., Agrawal, A., Liao, W.K., Manne, F. and Choudhary, A., 2012, November. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (p. 62). IEEE Computer Society Press.

[4] He, Y., Tan, H., Luo, W., Mao, H., Ma, D., Feng, S. and Fan, J., 2011, December. Mr-dbscan: an efficient parallel density-based clustering algorithm using mapreduce. In 2011 IEEE 17th International Conference on Parallel and Distributed Systems (pp. 473-480). IEEE.

[5] Lulli, A., Dell'Amico, M., Michiardi, P. and Ricci, L., 2016. NG-DBSCAN: scalable density-based clustering for arbitrary data. Proceedings of the VLDB Endowment, 10(3), pp.157-168.

[6]Song, H. and Lee, J.G., 2018, May. RP-DBSCAN: A superfast parallel DBSCAN algorithm based on random partitioning. In Proceedings of the 2018 International Conference on Management of Data (pp. 1173-1187). ACM.