Performance of Big Data versus High-Performance Computing: Some Observations

Helmut Neukirchen

School of Engineering and Natural Sciences, University of Iceland, Reykjavík, Iceland
Email: helmut@hi.is

Abstract. The two prevalent paradigms for parallel processing are HPC and the newer big data platforms. In addition to comparing their general properties, a survey and run-time comparison of implementations of the DBSCAN clustering algorithm for these two paradigms are provided.

Keywords: Big Data, High-Performance Computing, Benchmarking

1 Introduction

Computationally intensive simulations require parallel processing. The standard technology for huge non-embarrassingly parallel, but rather tightly-coupled computational problems is *High-Performance Computing* (HPC). Highly praised contenders for huge parallel processing problems are big data processing frameworks such as Apache Hadoop or Apache Spark. Hence, they might be considered an alternative to HPC for distributed simulations. To be able to decide whether HPC or big data platforms are better suited for computationally intensive problems, this paper gives a brief overview on these two paradigms and their platforms and compares as main contribution their run-time performance and scalability using as examples different implementations of the *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN) clustering algorithm.

2 Paradigm shift: HPC versus Big Data

HPC is tailored to typically CPU-bound computationally expensive jobs. Hence, rather expensive hardware is used, e.g. compute nodes containing fast CPUs including many cores, very fast interconnects (e.g. InfiniBand) for communication between nodes, and centralized *Storage-Area Network* (SAN) storage. To make use of the many cores per CPU, shared-memory multi-threading based on *Open Multi-Processing* (OpenMP) is applied. To make use of the many nodes connected via the interconnects, an implementation of the *Message Passing Interface* (MPI) is used. The underlying programming model is low-level, but allows tightly-coupled parallel processing. Low-level, but fast programming languages such as C, C++ and Fortran are used.

The big data paradigm is tailored to process huge amounts of data, however the actual computations to be performed on this data are often not that computationally intensive. To achieve high-throughput, locality of data storage is exploited by using distributed file systems storing locally on each node a part of the data. The big data approach aims at doing computations on those

Table 1. Run-time (in Seconds) vs. Number of Cores for 3 704 351 points

Number of cores	1	2	4	8	16	32
HPDBSCAN	114	59	30	16	8	6
PDSDBSCAN	288	162	106	90	85	88
ELKI	997	–	–	–	–	–
RDD-DBSCAN	7 311	3 521	1 994	1 219	889	832

nodes where the data is locally available. An example is Apache Hadoop which, however, has the disadvantage that only the MapReduce paradigm is supported which restricts the possible class of parallel algorithms and in particular may lead to unnecessarily storing intermediate data on disk instead of allowing to keep it in fast RAM. This weakness is overcome by Apache Spark [2] which is based on *Resilient Distributed Dataset*s (RDDs) which are able to store a whole data set in RAM, distributed in partitions over the nodes of a cluster. While RDDs may be kept in RAM, required data may not be available in the local RDD partition of a node. In this case, it is necessary to re-distribute data between nodes. Such shuffle operations are expensive, because slow network transfers are needed for them. High-level programming languages such as Java, Scala or Python are used.

3 Run-time/Scalability of DBSCAN Implementations

To investigate the run-time performance and scalability of existing scientific libraries (needed to ease building parallel applications, such as simulations) and their underlying HPC or Spark platform, we surveyed and benchmarked parallel open-source implementations of the the clustering algorithm DBSCAN [4]. The underlying idea of DBSCAN is that for each data point, the neighbourhood within a given *eps* radius has to contain at least a *minpts* points to form a cluster, otherwise it is considered as noise. Depending on the implementation and the size of *eps* in comparison to the size of the whole data, the time complexity is $O(n \log n)$ to $O(n^2)$ with the latter leading to scalability problems for big data.

We found two open-source parallel DBSCAN implementations for HPC using C++: PDSDBSCAN. [7] and HPDBSCAN [5]). The highly-optimized serial Java implementation ELKI [9] was used as reference for the four parallel Scala/*Java Virtual Machine* (JVM)-based open-source implementations we found for Spark: Spark DBSCAN [6], RDD-DBSCAN [3], Spark_DBSCAN [1], and DBSCAN On Spark [8]. Results from running the first four implementations[1] are shown in Table 1. All measurements were performed on the same, identical cluster for HPC and Spark using 3 704 351 2D geo-tagged tweets: except HPDBSCAN, none of the implementations scaled well beyond 16 cores and in particular the RDD-DBSCAN implementation for Spark was significantly slower.

To investigate performance on a bigger dataset using more cores, we clustered 16 602 137 geo-tagged tweets on several hundred cores (Table 2). Again, the HPC implementation performed significantly better than the ones for Spark. As these were already with a high number of cores very slow (RDD-DBSCAN with

[1] For the latter three, no detailed experiments were made due to $O(n^2)$ complexity (RDD-DBSCAN), being already extremely slow with 928 cores (Spark_DBSCAN) and providing in fact only an approximation of DBSCAN (DBSCAN on Spark).

Table 2. Run-time (in Seconds) vs. Number of Cores for 16 602 137 points

Number of cores	1	384	768	928
HPBDBSCAN	2 079	10	8	–
ELKI	15 362	–	–	–
Spark_DBSCAN	–	–	–	Exception
RDD-DBSCAN	–	–	–	5 335

928 cores was only three times faster than ELKI using one core) or threw an exception, we did not perform measurements with a lower number of cores.

4 Conclusions

In summary, none of the DBSCAN implementations for Apache Spark is anywhere near to the HPC implementations. It can be speculated that this is because in HPC, parallelization needs to manually implemented and thus gets more attention in contrast to the high-level big data approaches where the developer gets not in touch with parallelization. Another reason to prefer HPC for compute-intensive simulations is that already based on the used programming languages, run-time performance of the JVM-based Spark platform can be expected to be one order of magnitude slower than C/C++ (compare ELKI vs. HPDBSCAN). While RDDs support a bigger class of non-embarrassingly parallel problems than MapReduce, Spark still does not support as tight-coupling as OpenMP and MPI used in HPC – which might however be required for simulations. On the other hand, due to the high-level programming languages such as Scala and the fact that Spark code can be written like serial code without having to care about parallelization, considerably less implementation efforts can be expected when using Spark. Also, in contrast to HPC where no fault tolerance is included (a single failure on one of the many cores will cause the whole HPC job to fail), the big data platforms have the advantage of being are fault-tolerant. Finally, the commodity hardware typically used as big data platform is cheaper.

References

1. aizook: Spark_DBSCAN source code. GitHub repository (2014), `https://github.com/aizook/SparkAI`
2. Apache Software Foundation: Apache Spark (2016), `http://spark.apache.org/`
3. Cordova, I., Moh, T.S.: DBSCAN on Resilient Distributed Datasets. In: 2015 Int. Conf. on High Performance Computing & Simulation (HPCS). IEEE (2015)
4. Ester, M., et al.: Density-based spatial clustering of applications with noise. In: Proc. of the 2nd Int. Conf. on Knowl. Discovery and Data Mining. AAAI (1996)
5. Götz, M., et al.: HPDBSCAN: highly parallel DBSCAN. In: Proc. of the Workshop on Machine Learning in High-Performance Computing Environments. ACM (2015)
6. Litouka, A.: Spark DBSCAN source code. GitHub repository (2014), `https://github.com/alitouka/spark_dbscan`
7. Patwary, M.M.A., et al.: A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In: Supercomputing (SC2012). IEEE (2012)
8. Raad, M.: DBSCAN On Spark source code. GitHub repository (2016), `https://github.com/mraad/dbscan-spark`
9. Schubert, E., et al.: A framework for clustering uncertain data. PVLDB 8(12), 1976–1979 (2015)