VariantDBSCAN: Maximizing Clustering Throughput

Mike Gowanlock
MIT Haystack Observatory
gowanloc@mit.edu

Contents

1 Introduction

DBSCAN [1] is a popular algorithm for clustering spatial data. The algorithm takes as input two parameters: ϵ and *minpts*, which define a neighborhood radius and density threshold to determine if points belong in a cluster. The algorithm works by chaining points together, thus making it good for detecting clusters of arbitrary shape. Futhermore, it does not output a predefined number of clusters, making it an unsupervised approach.

Given that researchers typically need to execute DBSCAN multiple times with different parameters, VARIANTDBSCAN uses multiple optimizations to improve parallel clustering

throughput. In short, a variant is the definition of the two parameters that are taken as input into DBSCAN. VARIANTDBSCAN exploits data reuse between variants, reduces memory access overhead through efficient indexing of the points in the dataset, and schedules threads to variants to maximize data reuse between variants. See our paper [2] for more information about the optimizations. This code is a production version and does not contain all of the scheduling heuristics and cluster reuse approaches as described in [2], as we selected the optimizations that performed the best in our experimental evaluation to limit the number of configuration options and parameters required to install and run VARIANTDBSCAN.

2 Installation

We have included both C/C++ and Python interfaces to run VARIANTDBSCAN. Both of these use a common shared library developed in C++ that must be compiled on the system before use. For the python installation, there are two options, either install manually or use the pip package manager. A roadmap of instructions is given below depending on your preferred installation.

- C/C++: All instructions in Section 2.1, then Section 2.2.
- Python (manual): All instructions in Section 2.1, then Section 2.3.
- Python (pip install): Make sure you meet the requirements in Section 2.1.1, then Section 2.4.

2.1 Shared Library

2.1.1 Requirements

The shared library, `libSharedVDBSCAN.so`, has been compiled and run using standard libraries, compilers and Linux operating system as outlined below. We have included the versions of the software we used when compiling the shared library.

For Mac OSX users, the code has been compiled using the python manual installation under OSX v.10.12. However, you must install g++ and OpenMP. At the time of writing, OSX v.10.12 and potentially other versions do not support LLVM/OpenMP.

1. Ubuntu (14.04.3)
2. GNU g++ (4.8.4)
3. OpenMP for multithreading (3.1, part of g++ installation)

2.1.2 Compile Options

There are two configuration options that can be modified in the `makefile`:

1. `SEARCHTHREADS` Default value: 4.
   This should be set to some number up to the number of cores on your machine. This is the maximum number of threads that will concurrently cluster the variants.

2. `libpath` Default value: `/usr/local/lib/`
   This is the location of the shared libraries on your system. When compiling the shared library, the `makefile` will copy the library to this location.

2.1.3 Installation

To install the shared library, simply navigate to the src directory and type the following: `sudo make libSharedVDBSCAN.so`

Note: you need to use `sudo` so the `makefile` can copy the library into `libpath`. However, if you do not have permissions, `libSharedVDBSCAN.so` will be in the src directory, and you can link to it as you see fit.

2.2 C++ Interface

In the `src` directory, the code in `c_test_prog.cpp` is an example test program that uses the shared library. To compile this program, type: `make c_test_prog`. This assumes that `libSharedVDBSCAN.so` has been successfully copied into `libpath`. Otherwise, modify `makefile` to link to the location of `libSharedVDBSCAN.so`.

2.3 Python Interface: Manual Installation

In the `src` directory open `variantdbscan.py` and modify `lib_path` to be the directory that contains `libSharedVDBSCAN.so`. The python wrapper, `variantdbscan.py`, links to the shared library. The following code in `variantdbscan.py` should be modified according to the comments:

```
# Get library path
# Use the following line if installed by pip
lib_path = os.path.dirname(os.path.realpath(__file__)) + '../'
# Otherwise, comment out the above line and
# uncomment the following line, adjusting path as necessary
# lib_path = '/usr/local/lib/'
```

2.4 Python Interface: Using the Pip Package Manager

To install via pip, the environment variable `NTHREADS` variable should be set to some number up to the number of cores on your machine. This is the maximum number of threads that will concurrently cluster the variants. If you do not set `NTHREADS`, it will default to 4.

To install, type: `sudo NTHREADS=4 pip install vdbscan`

3 Usage

The shared library is called `libVDBSCAN()`. The function signature is as follows:

```
int libVDBSCAN(double * inputx, double * inputy, unsigned int datasetSize,
double * inputEpsilon, unsigned int * inputMinpts, unsigned int numVariants,
```

`int MBBsize, unsigned int * retArr, bool verbose);`

Explanation of the variables:

- `double * inputx` and `double * inputy` - Dataset values in two dimensions as two arrays.
- `unsigned int datasetSize` - Number of points in the dataset (size of the length of `inputx` and `inputy`).
- `double * inputEpsilon` and `unsigned int * inputMinpts` - Arrays of variant ϵ and *minpts* values.
- `unsigned int numVariants` - The number of variants to be clustered (size of the length of `inputEpsilon` and `inputMinpts`).
- `int MBBsize` - The number of points per minimum bounding box for the R-tree index (Default value: 70).
- `unsigned int * retArr` - An array containing the result set. Each point in the dataset is assigned a cluster id. This array is the cluster ids of points for each variant. The length of this array is: `datasetSize` $\times$ `numVariants` (or $|D| \times |V|$ in [2]).
- `bool verbose` - Flag to turn on or off optional output.

`retArr` contains the results across all variants. The cluster id is given for each point, where an id of 0 indicates that it is a noise point (not in a cluster).

3.1 C++

First, you must compile `c_test_prog.cpp` outlined in Section 2.2. Here, we go through the example clustering in the test program.

The following variables are set to cluster two variants ($\epsilon = 1.0$, $minpts = 10$) and ($\epsilon = 0.5$, $minpts = 12$).

- `dataset` – set to `test_dataset.csv`
- `numVariants` – set to 2
- `inputEps` – set to [1.0, 0.5]
- `inputMinpts` – set to [10, 12]

Run the program: `./c_test_prog`

The program will output the results, as shown in Appendix A. A plot of the results of both variants is also shown.

3.2 Python

The python test program in the `src` directory is called `variant_example.py`. First, depending on how the VariantDbscan interface was installed (manually or through pip), the correct import statement should be selected:
When using the pip installed version:
```
from vdbscan import VariantDBSCAN
```

When using the manually installed version:
```
from variantdbscan import VariantDBSCAN
```

The following variables are set to cluster two variants ($\epsilon = 1.0$, $minpts = 10$) and ($\epsilon = 0.5$, $minpts = 12$).

- `data = pd.read_csv("test_dataset.csv", header=None,names=['x','y'])`
- `eps = np.array([1, 0.5])`
- `mp = np.array([10, 12])`

Run the program: python `variant_example.py`

The program will output the results, as shown in Appendix A. A plot of the results of both variants is also shown.

4 Source Code Overview

While this document is intended primarily for those wanting to use the source code for clustering, we outline the structure of the code. The reader intending to make algorithmic changes to the code should first be very familiar with the Dbscan algorithm and then the VariantDbscan paper [2]. See the doxygen directory for the html and pdf versions of the code. The callgraph for `libVDBSCAN` is probably the most useful.

The flow of the program `libVDBSCAN` (in `main.cpp`) is outlined as follows. First, the variants are imported and marshalled into a vector. This vector is sorted based on the Greedy scheduling heuristic. Next, the dataset is imported from the arrays into a vector. The data is then binned spatially such that data points nearby each other are close to each other in memory, thus improving locality, and allowing for the R-tree to index points together in compact MBBs. The R-tree index is initialized, where the leaf nodes are defined by MBBs that contain multiple points. To allow for the individual data points to be accessed from the R-tree, a lookup array is used. The MBBs that represent multiple points are inserted into the R-tree. Next, another R-tree index is initialized that indexes each point in its own MBB. This index is not used for ϵ-neighborhood searches, rather it is used in the process of expanding a cluster to reuse data from a previously executed variant. The DBScan objects are initialized, where each object corresponds to a single variant. The scheduler is initialized which creates a job queue to assign variants to threads in the Greedy scheduling order. Furthermore, the scheduler determines whether a completed variant can be used as input to cluster a variant yet to be processed.

The implementation uses OpenMP. A thread executes a single loop iteration, where the thread is assigned a variant to process by the scheduler. Also, the scheduler determines if a given variant assigned to a thread can reuse any completed variants. Two execution pathways can occur: 1) If there are no variants that can be reused for a given variant, then the variant is clustered from scratch; or 2) if the variant can reuse the results from a completed variant, then it will use the previous cluster assignments as input into the algorithm. Thus, a single thread executes a single variant (whether clustering from scratch or reusing data from a previous variant). After all of the variants have been clustered, the result set array is returned. To return the data, two transformations have to occur. First, the dataset needs to be mapped to its original ordering (recall that we reorder the data to improve locality and for indexing purposes). Secondly, the variant results (assignment of points to clusters) need to be reordered to be the same order that was sent to the function.

The source code contains many comments to help the reader understand the program. The code has three major classes that will be described in the following subsections.

- RTree – A spatial index to efficiently find the data points in the dataset, primarily for ϵ-neighborhood searches.
- schedule – Determines the ordering in which the variants are executed.
- DBScan – Each DBScan object executes a single variant.

4.1 R-Tree

The R-Tree is a spatial index used to find the data points in the dataset (ϵ-neighborhood searches) and is used in the process of reusing data from the output of one variant as input into another.

A single R-tree is constructed that contains a number of minimum bounding boxes (MBB) that contain points spatially nearby each other in the dataset. The number of points per MBB is set by `MBBsize`, described in Section 3. Putting multiple points in a single MBB reduces tree depth at the expense of more filtering overhead. When clustering, this R-tree is traversed by multiple threads to perform the ϵ-neighborhood searches in the algorithm for different variants.

Another R-tree is constructed that indexes each point in its own MBB. This is used when taking the clusters from the output of one variant, and using it as input to "short circuit" the clustering of another variant. That is, the data points belonging to one cluster may be common to two variants and thus expensive ϵ-neighborhood searches can be avoided. See [2] for more information.

Pointers to two R-tree indexes are taken as input to DBScan objects.

4.2 Scheduling

To allow for data reuse between variants, the output of one variant contains cluster assignments that may apply to another variant. In short, clusters with smaller values of ϵ can be used as seeds to create clusters that have higher values of ϵ. In order to maximize the chances of data reuse between variants, the scheduling algorithm creates an ordering of the execution

of variants as a function of the ϵ and *minpts* values. This is conflated with the number of threads used. If variants are still waiting to execute in the work queue and a thread is idle, the thread will be assigned a variant to process where the two execution pathways can occur (cluster from scratch or reuse data). The Greedy schedule in [2] is used.

4.3 DBScan

Initializing the object takes as input: the dataset, the variant parameter values: ϵ and *minpts*, an R-tree for ϵ-neighborhood searches, an accompanying lookup array into the dataset, and a high resolution R-tree which is utilized when expanding a cluster when reusing data between variants. As mentioned above, two execution pathways can occur: 1) If there are no variants that can be reused for a given variant, then the variant is clustered from scratch; or 2) if the variant can reuse the results from a completed variant, then it will use the previous cluster assignments as input into the algorithm.

If the variant is clustered from scratch, then the class executes the function `algDBScanParallel()`. This is the standard DBSCAN algorithm with the index optimization that uses an R-tree with multiple points per MBB. Note that a single thread executes a single variant, and multiple threads concurrently cluster different variants.

If the variant can reuse data from a previously completed variant, then `assignPointsToPredefinedCluster()` is executed. This method takes as input the id of the previously finished variant object. Through a pointer to the object, the variant to be processed has access to the cluster assignments of all of the points in the completed variant. With this information, clusters can be reused as the points belonging to one cluster are guaranteed to be common in another cluster (based on the criteria outlined in [2]). `assignPointsToPredefinedCluster()` has a few major steps, as follows:

1. A list of clusters is obtained from the previously completed variant.
2. These clusters are prioritized using the density cluster reuse heuristic.
3. The list of candidate clusters to expand are expanded by adding new points to the cluster with ϵ-neighborhood searches. This continues until all of the points in the candidate clusters have been visited or the candidate clusters have been destroyed through the process of reuse.
4. Finally, any points in the dataset that have not been visited, are clustered (they may have been noise points in the previously completed variant, but may form a cluster in the new variant).

ACKNOWLEDGMENTS

We acknowledge support from NSF ACI-1442997.

A Example Output

Data is in the format: point id, x value, y value, cluster assignment. Cluster id 0 is a noise point. The dataset, `test_dataset.csv` has 225 points. Plots of the assignment of points to clusters is shown in Figures 1 and 2, where colors indicate different clusters.

Output:

Variant: 0, Epsilon: 1.000000, minpts: 10
0, 6.689900, -0.131300, 2
1, -3.989800, -1.113700, 1
2, 4.268700, -0.687100, 2
3, -6.031600, 1.570700, 1
4, 4.704300, -0.315900, 2
5, -3.534000, -1.283700, 1
6, -6.767100, 1.524200, 1
7, 3.293400, -2.245800, 2
8, -5.150900, -0.269400, 1
9, -4.670700, -0.986800, 1
10, 5.630800, 0.360500, 2
...
215, 3.758700, 5.909100, 0
216, -5.874500, 3.702000, 0
217, -3.680800, -0.943700, 1
218, -4.298700, -0.441000, 1
219, 3.337700, -2.005900, 2
220, -6.111300, 1.477300, 1
221, 5.133000, 0.894600, 2
222, 6.219200, 0.936800, 2
223, -4.717400, -0.258900, 1
224, 6.067300, 0.611400, 2

Variant: 1, Epsilon: 0.500000, minpts: 12
0, 6.689900, -0.131300, 0
1, -3.989800, -1.113700, 1
2, 4.268700, -0.687100, 2
3, -6.031600, 1.570700, 1
4, 4.704300, -0.315900, 2
5, -3.534000, -1.283700, 1
6, -6.767100, 1.524200, 0
7, 3.293400, -2.245800, 0
8, -5.150900, -0.269400, 1
9, -4.670700, -0.986800, 1
10, 5.630800, 0.360500, 2
...
215, 3.758700, 5.909100, 0
216, -5.874500, 3.702000, 0

217, -3.680800, -0.943700, 1
218, -4.298700, -0.441000, 1
219, 3.337700, -2.005900, 0
220, -6.111300, 1.477300, 1
221, 5.133000, 0.894600, 2
222, 6.219200, 0.936800, 2
223, -4.717400, -0.258900, 1
224, 6.067300, 0.611400, 2

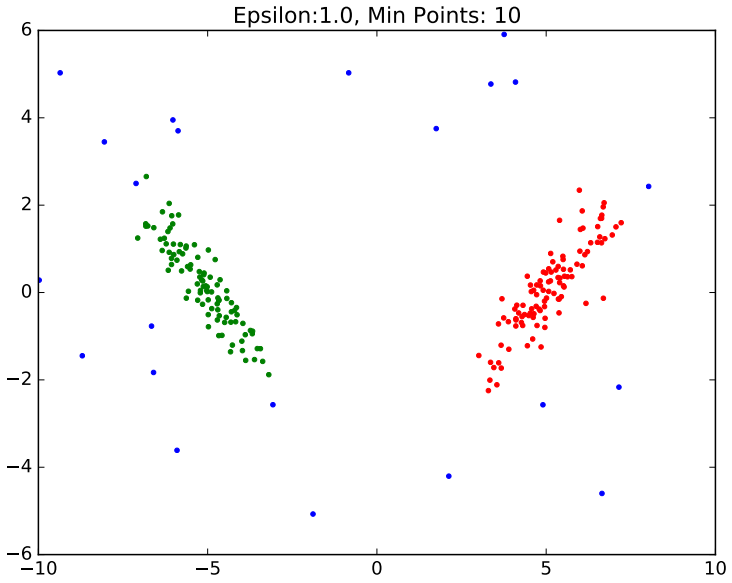

Figure 1: Plot of the clusters in the dataset with $\epsilon = 1.0$ and $minpts = 10$. Two clusters are detected (green, red) and noise points are blue.

References

[1] Martin Ester, Hans Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd KDD*, pages 226–231, 1996.

[2] M. Gowanlock, D. M. Blair, and V. Pankratius. Exploiting Variant-Based Parallelism for Data Mining of Space Weather Phenomena. In *Proceedings of the 30th IEEE International Parallel & Distributed Processing Symposium*, pages 760–769, 2016.

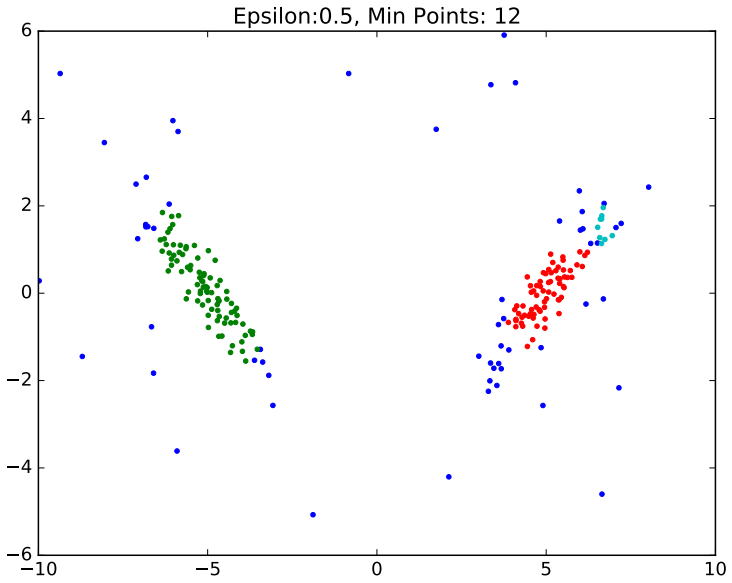

Figure 2: Plot of the clusters in the dataset with $\epsilon = 0.5$ and $minpts = 12$. Three clusters are detected (green, red, cyan) and noise points are blue.