# Supercomputer or Cloud Computing?

## Large-scale, High-computation Data Clustering Tasks with PyOMP on single machine and with PySpark on AWS EC2 instances

Jhao-Ting Chen (jhaoting), Wei-Lun Chiu (weilunc)
Carnegie Mellon University – 15-618 Parallel Computer Architecture and Programming – Final Project

Project Website

## Abstract

We compared the performance and costs of executing data clustering algorithms on a multi-core machine with PyOMP and on a set of workers with comparable resource constraints of AWS instances with PySpark. We analyzed the serial, parallelized, and distributed implementation of two clustering algorithms: K-means and DBSCAN, evaluated their trade-off between performance, financial cost, and different algorithmic strategies.

## PyOMP

PyOMP, released by Intel Corp., provides an easy-to-use API that allows developers to parallelize their Python code with minimal modifications, by adding OpenMP directives to their code. Built upon Numba, PyOMP offers users an OpenMP-similar syntax for accessibility. Programs written in C with OpenMP performs only 2.8% faster than programs in PyOMP. Numba is just-in-time (JIT) compiler that translates Python into optimized machine code at runtime, just before execution, providing significant speedups for computationally intensive tasks. Numba uses LLVM as the backend for its JIT compiler, capable of many optimization techniques including vectorization, automatic parallelization, and loop unrolling. Example PyOMP operations are shown in Table 1.

| PyOMP | OpenMP |
|---|---|
| `with openmp("parallel"):` | `#pragma omp parallel` |
| `with openmp("for"):` | `#pragma omp parallel for` |
| `with openmp("single"):` | `#pragma omp single` |
| `with openmp("task"):` | `#pragma omp task` |
| `with openmp("taskwait"):` | `#pragma omp taskwait` |
| `with openmp("barrier"):` | `#pragma omp barrier` |
| `with openmp("critical"):` | `#pragma omp critical` |
| `with openmp("for schedule(static [,chunk])"):` | `#pragma openmp for schedule(static, [chunk])` |
| `with openmp("for reduction(op:List)"):` | `#pragma omp for reduction(op:List)` |
| `with openmp("for private(List)"):` | `#pragma omp for private(List)` |
| `with openmp("for shared(List)"):` | `#pragma omp for shared(List)` |

Table 1: PyOMP supported operations and references to OpenMP

## PySpark

PySpark is a Python API for Apache Spark, a distributed computing system used for big data processing and analytics. Spark is designed to work with large-scale data processing tasks that require high-speed data processing and distributed computing capabilities. PySpark provides even more easy-to-use interface for users to handle data analytic tasks in Python.

## Results - PySpark

### Hardware Setup

We leveraged AWS EC2. t3.large. uses Intel Xeon Platinum 8000 series CPU with 2 threads per core and clock speed up to 3.0 GHz. Our experiment showed the maximum clock speed for each executor is 2.5GHz, and the same program runs two times longer on than the AMD 5600x. For comparison between PyOMP, we took 1 node (with 2 vCPU/threads) as 1 thread in AMD Ryzen5 5600x. The spot instance price of t3.large is 0.0637 USD/hr on April 16th 2023.

### Runtime v. nThreads

We set up each experiments where $nThread$ is the number of nodes in a cluster. The dataset were partitioned into $nThread * 2$ since each node has 2 threads, each with half the clock speed of AMD 5600x. There is a drastic time difference between PySpark and PyOMP, where a 50 seconds runtime for PyOMP with 12 threads requires **24 minutes** for PySpark with 12 nodes. The execution time of k-means with PySpark decreased linearly with number of executors. The runtime includes execution, synchronization, and Spark overheads, which is why the task took this long.

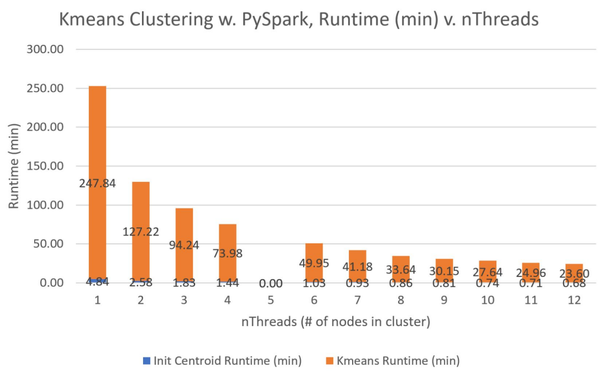



Figure 9: PySpark K-means total runtime w.r.t. nThreads

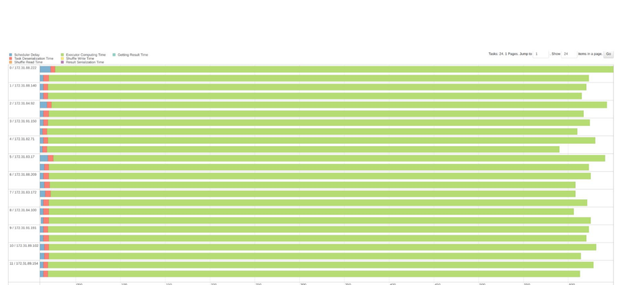
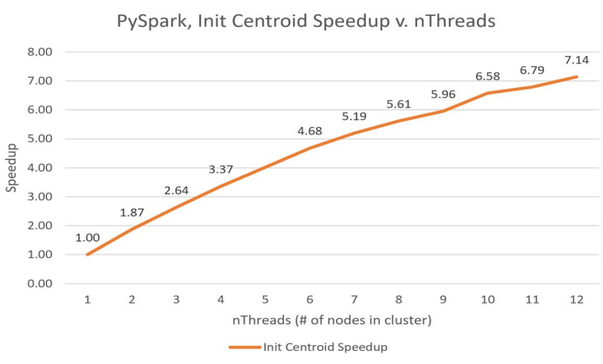
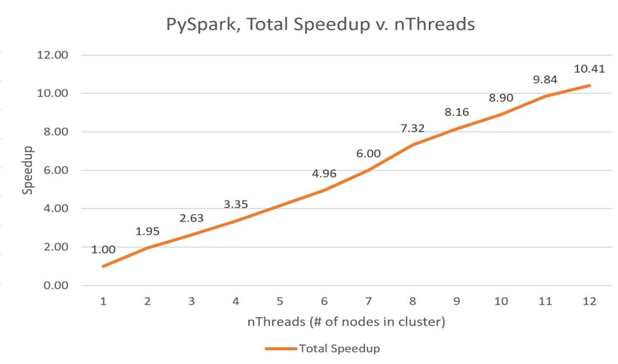


Figure 8: PySpark job executed on 12 executors




(a) Init Centroid Stage Speedup




(b) Total Speedup

Figure 10: PySpark K-means runtime w.r.t. nThreads

## Conclusion - PySpark

Our implementation of k-means clustering parallelized with PySpark achieved **10.41x speedup** on a 12-node cluster, with each node having 2 vCPUs, each having half of the computation power of AMD Ryzen5 5600x. The speedup increased linearly with respect to the number of node executing. However, due to the heavy overhead of Spark job and synchronization across nodes, the runtime is drastically slower than PyOMP implementation of the same code.

**DBSCAN Results are on Project Website: jtchen0528/cmu-15618-project**

## Results – PyOMP

### Hardware Setup

We ran the experiments on one of our personal computer with AMD Ryzen5 5600x as CPU. AMD 5600x has 6 cores (12 threads), maximum clock speed of 4.6 GHz. The pricing of AMD Ryzen5 5600x is 299 $USD.

### Serial Implementation Analysis

| Clusters($k$) | 64 | | 36 | | 10 | |
|---|---|---|---|---|---|---|
| Implementation | Serial | PyOMP | Serial | PyOMP | Serial | PyOMP |
| Iterations | 100 | 100 | 255 | 255 | 92 | 92 |
| Accuracy | 83.74% | 83.74% | 77.73% | 77.73% | 59.42% | 59.42% |
| Homogeneity | 77.62% | 77.62% | 71.40% | 71.40% | 50.88% | 50.88% |

Table 2: Correctness of serial and PyOMP implementation of K-means clustering

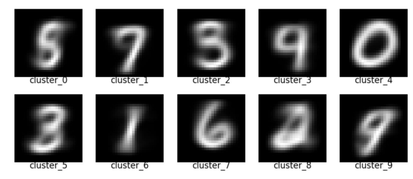



Figure 2: k-means clustered centroids with $k = 10$

## Runtime v. nThreads

The runtime for the three stages in k-means are shown in Fig 3. Clustering Data stage accounts for most of the runtime. This is expected since clustering data points involves heavy computation on Euclidean distances.

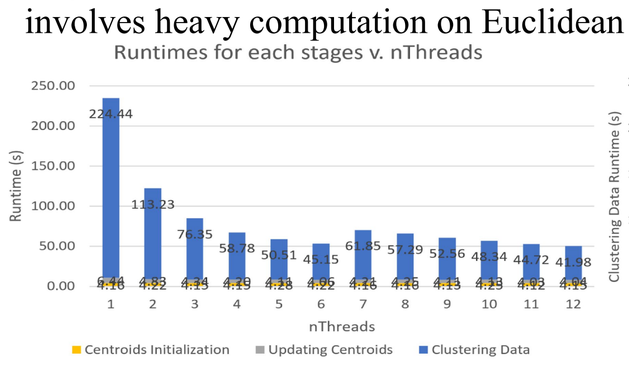



Figure 3: PyOMP K-means total runtime w.r.t. nThreads

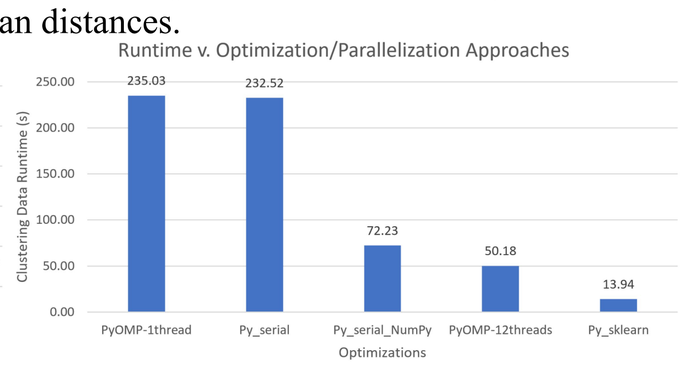


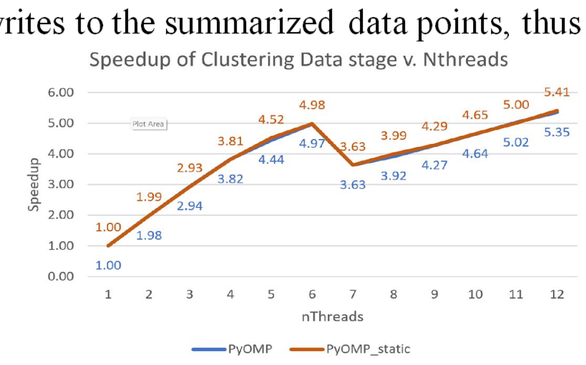

Figure 5: PyOMP K-means runtime with different optimization approaches

## Clustering Data Analysis

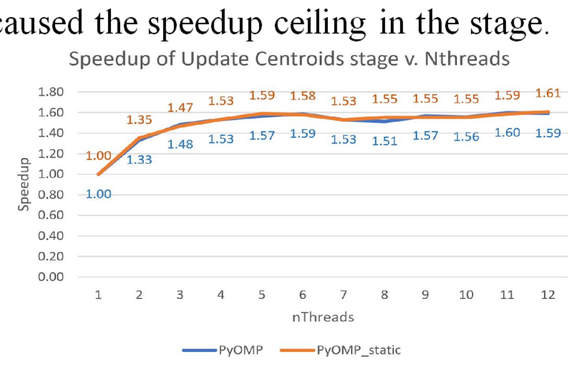
In Fig 4a, from $nThread = 1$ to 6, the speedups increased almost linearly and reached **4.98x**. An interesting phenomenon occurred when $nThread = 7$, the performance decreased but later increased to **5.41x** with $nThread$ increased to 12. The reason is that the chip has 6 cores, each with 2 threads. Synchronizing and communicating across the cores had less overheads. Threads that shares the resource in a core has more communication overheads and contentions.

### Atomic Action Analysis

Fig 4b illustrated the speedups of Updating Centroids stage w.r.t. the number of threads executing. The atomic actions speedups were limited at around **1.6x** when $nThread$ increases. The atomic operations in each threads contented for execution of writes to the summarized data points, thus caused the speedup ceiling in the stage.




(a) Clustering Data stage




(b) Updating Centroids stage

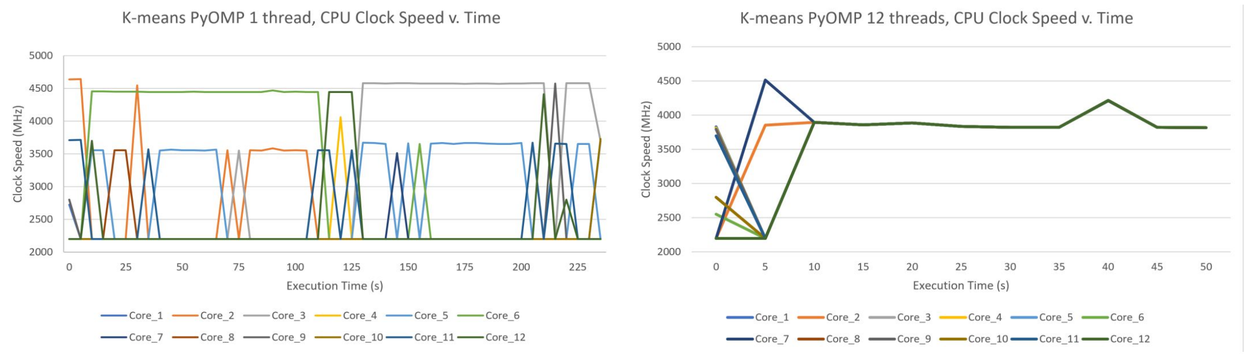
Figure 4: PyOMP K-means stage speedup w.r.t. nThreads

## Different Optimization Approaches

In Fig 5, we see that the same program but optimized with NumPy operations achieves a speedup comparing with plain implementation. Our PyOMP implementation with 12 threads speedup out-performed the NumPy-optimized program by **1.44x**. The fastest scikit-learn implements has more optimized K-means algorithm with built-in OpenMP support on multi-threading.

## CPU Utilization Analysis

To better inspect the threads utilization during execution, we logged the CPU clock rate. For PyOMP parallelized implementation, some logs are shown in Fig 6. As demonstrated in Fig 6a, there is always 1 core with the clock speed at about 4.5 GHz due to context swtiching. Fig 6b demonstrated the cores were executing exact same computation.




(a) PyOMP with 1 thread

(b) PyOMP with 12 threads

Figure 6: PyOMP K-means CPU Utilizations

## Conclusion – PyOMP

Our implementation of k-means clustering parallelized with PyOMP achieved **5.41x speedup on a 6 core, 12 threads CPU**; and **4.98x speedup with 6 cores**. The speedup **out-performed the NumPy-optimized implementation by 1.44x**. We examined the maximum speedup of **1.6x for atomic operations**. For AMD Ryzen5 5600x, the speedup with over 6 threads declined due to resource sharing within a core. We inspected the execution of PyOMP on switching context with CPU clock speed.

## Conclusion

The same workload executed 50.18s with a 299.00 $USD AMD Ryzen5 5600x, whereas executed 1440s with 13 (with driver node) AWS EC2 t3.large instance total of 0.33 $USD. The two use cases has its suitable scenario, it's a trade-off between speed and resource constraints.

# From Python to C++: A Journey to Efficient DBSCAN

Carnegie Mellon University

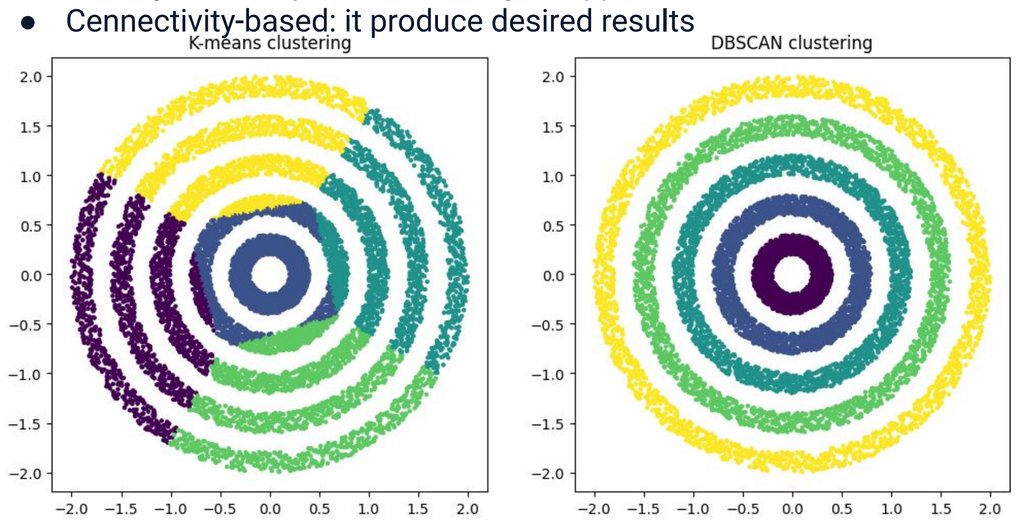
Student: Wei-Lun Chiu, Jhao-Ting Chen

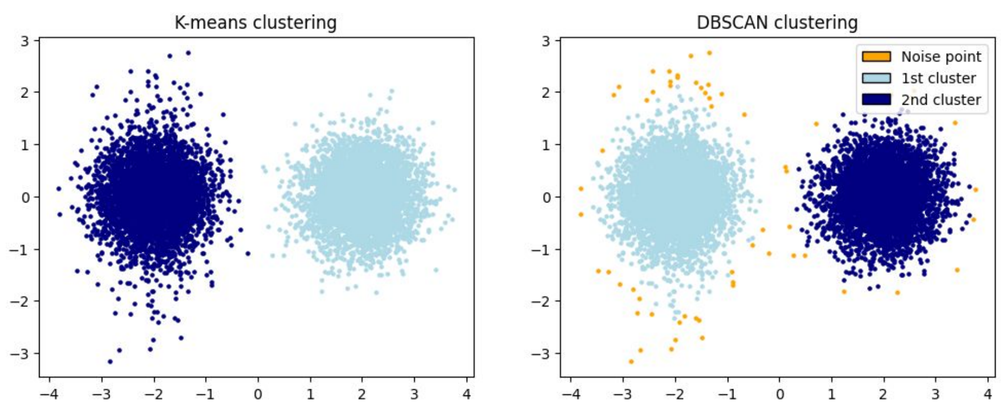
Carnegie Mellon University, Pittsburgh, PA

Electrical & Computer ENGINEERING

## DBSCAN: What, Why, How?

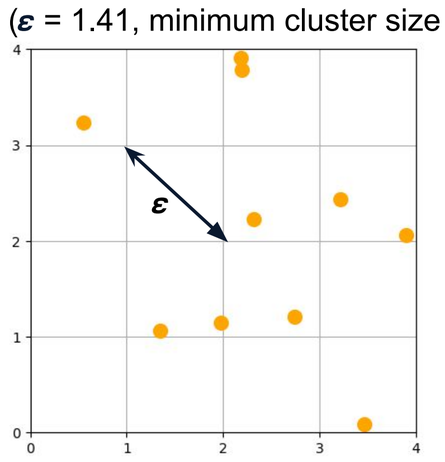
- **D**ensity-**B**ased **S**patial **C**lustering of **A**pplications with **N**oise
- Cennectivity-based: it produce desired results



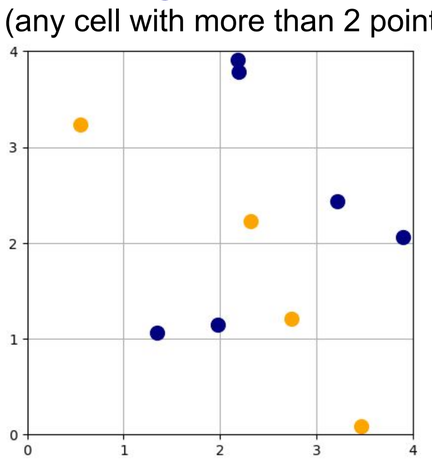




- Grid-based DBSCAN Visualization (***ε*** is minimum connectivity threshold)

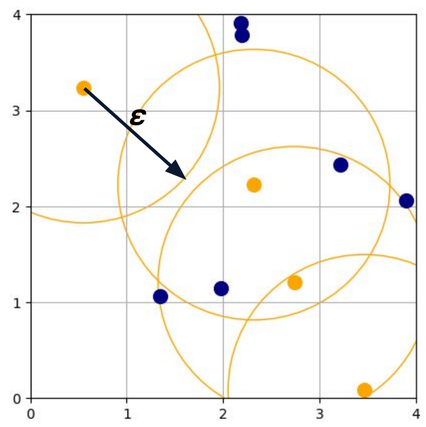
1. Initial Points
(***ε*** = 1.41, minimum cluster size = 2)



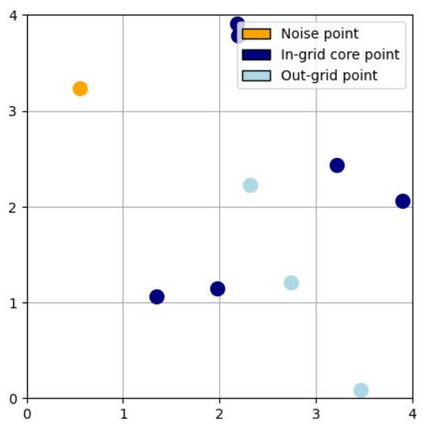

2. Mark **In-grid Core Cells**
(any cell with more than 2 points)



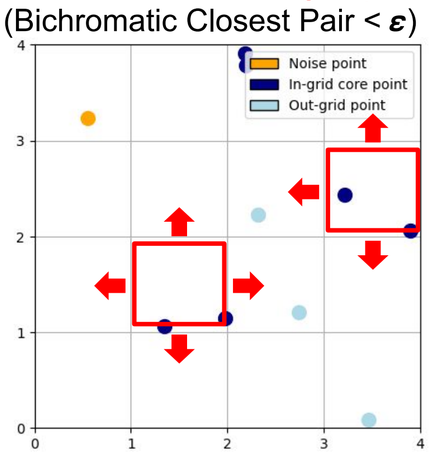
3. Mark Out-grid Core Cells
(any non-core cell contain a point connects to more than 1 points)



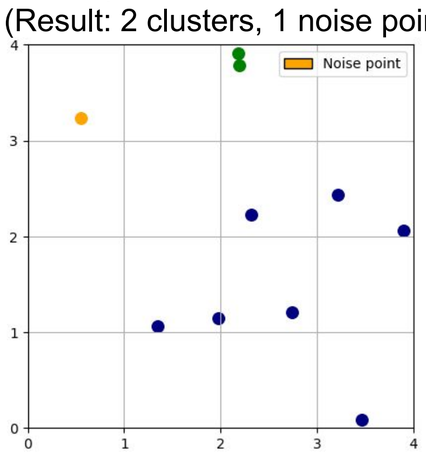

4. Finalized Point Classification
(core point and noise point)




5. Core Cell Clustering Expansion
(Bichromatic Closest Pair < ***ε***)




6. Complete!
(Result: 2 clusters, 1 noise points)




## Poor Parallel Performance in Python


## Analysis of Python Parallelism

- The main obstacle: Global Interpreter Lock (GIL)
  - No multi-thread parallelism allowed!
- PyOMP
  - Generate machine code. Super-fast (near C-level performance).
  - Restrictions:
    - No dynamic array, set, etc. No complex data structure
    - No recursive. No incomplete type. (No tree allowed!)
- Multiprocessing Module
  - Generates processes. Each process has its own GIL.
  - Too much overhead, too slow. Only 1.7x speedup with 6 threads.

## Analyzing Serial Runtimes in C++

| Procedure | Runtime(s) | Percentage |
|---|---|---|
| Gridify | 0.011691 | 0.9538% |
| Mark in-grid cores | $1.68 * 10^{-6}$ | 0.0001% |
| Mark out-grid cores | 0.00318535 | 0.2599% |
| **Expand** | 1.21085 | **98.7861%** |

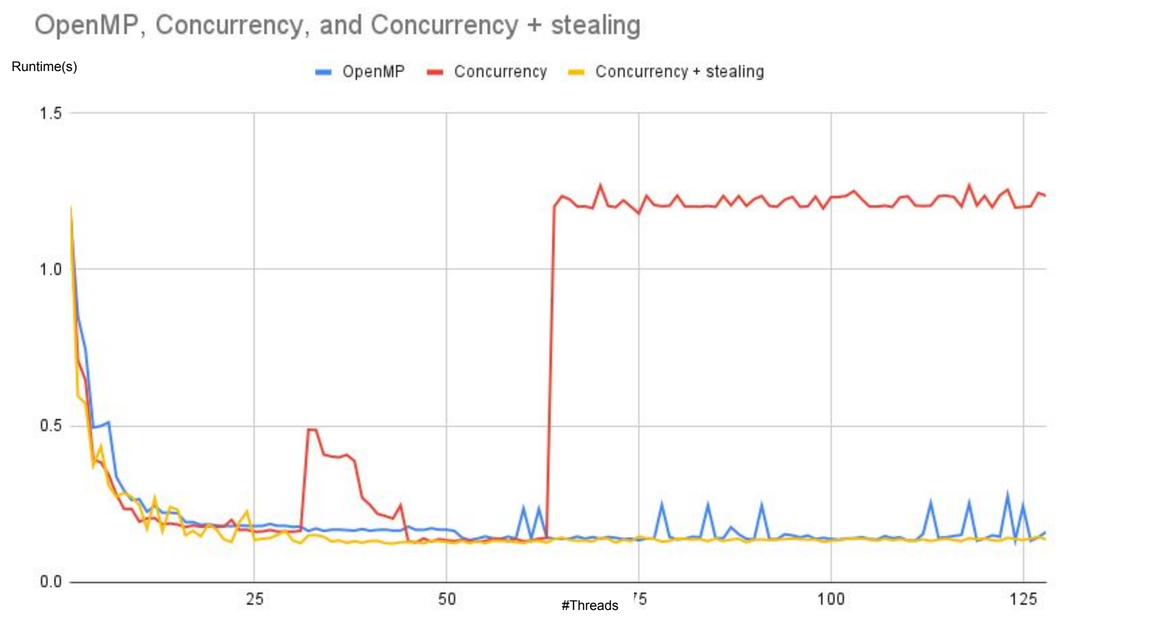
Table 1: Runtime and percentage breakdown for each step of the procedure

## C++ Concurrency API for Parallelism

- Part of C++ Standard Library since C++11
- Parallel implementation of expand in grid-based DBSCAN using **lock-free** UNION-FIND data structure for neighboring cell connectivity.

## Workload Balancing with Work-Stealing

- Observed workload imbalance (**Red line** in the following figure)
- Implement a work-stealing mechanism to balance workload (**yellow line**)
  - Thread-Specific work queues and **fine-grained locks**
  - Non-busy threads attempt to steal work from `(treadID+1)%nthreads`.

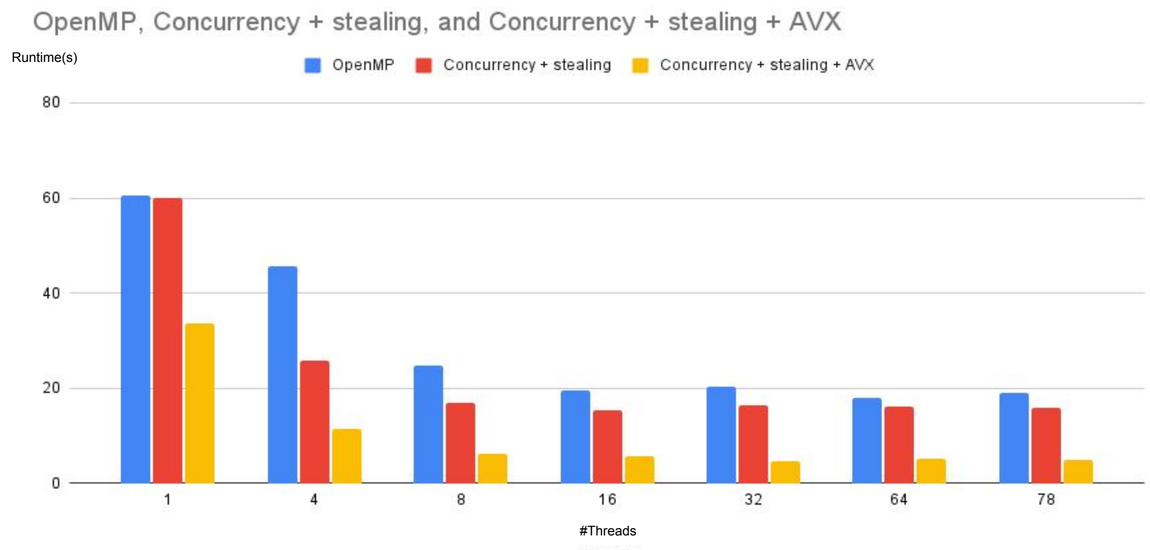



## Performance Analysis with Perf

- Memory allocation/release (45%)
- Floating point arithmetic (28%)

## Floating Point Optimization with AVX

- Optimize floating point arithmetic with **AVX** instructions to achieve **12.79x** speedup.




| #Ts | OpenMP | Concurrency + steal | Con. + steal + AVX |
|---|---|---|---|
| 1 | 60.5908 (0.99x) | 59.9386 (1.00x) | 33.683 (1.78x) |
| 4 | 45.5341 (1.32x) | 25.7059 (2.33x) | 11.5379 (5.20x) |
| 8 | 24.6547 (2.43x) | 16.8995 (3.55x) | 6.08641 (9.86x) |
| 16 | 19.4427 (3.09x) | 15.2388 (3.94x) | 5.57829 (10.76x) |
| 32 | 20.3203 (2.95x) | 16.4425 (3.65x) | 4.69244 **(12.79x)** |
| 64 | 17.8841 (3.35x) | 16.1772 (3.71x) | 5.05127 (11.88x) |
| 78 | 19.089 (3.14x) | 15.9645 (3.76x) | 4.80074 (12.50x) |

Table 2: Execution times (in seconds) for various thread counts and parallelization strategies