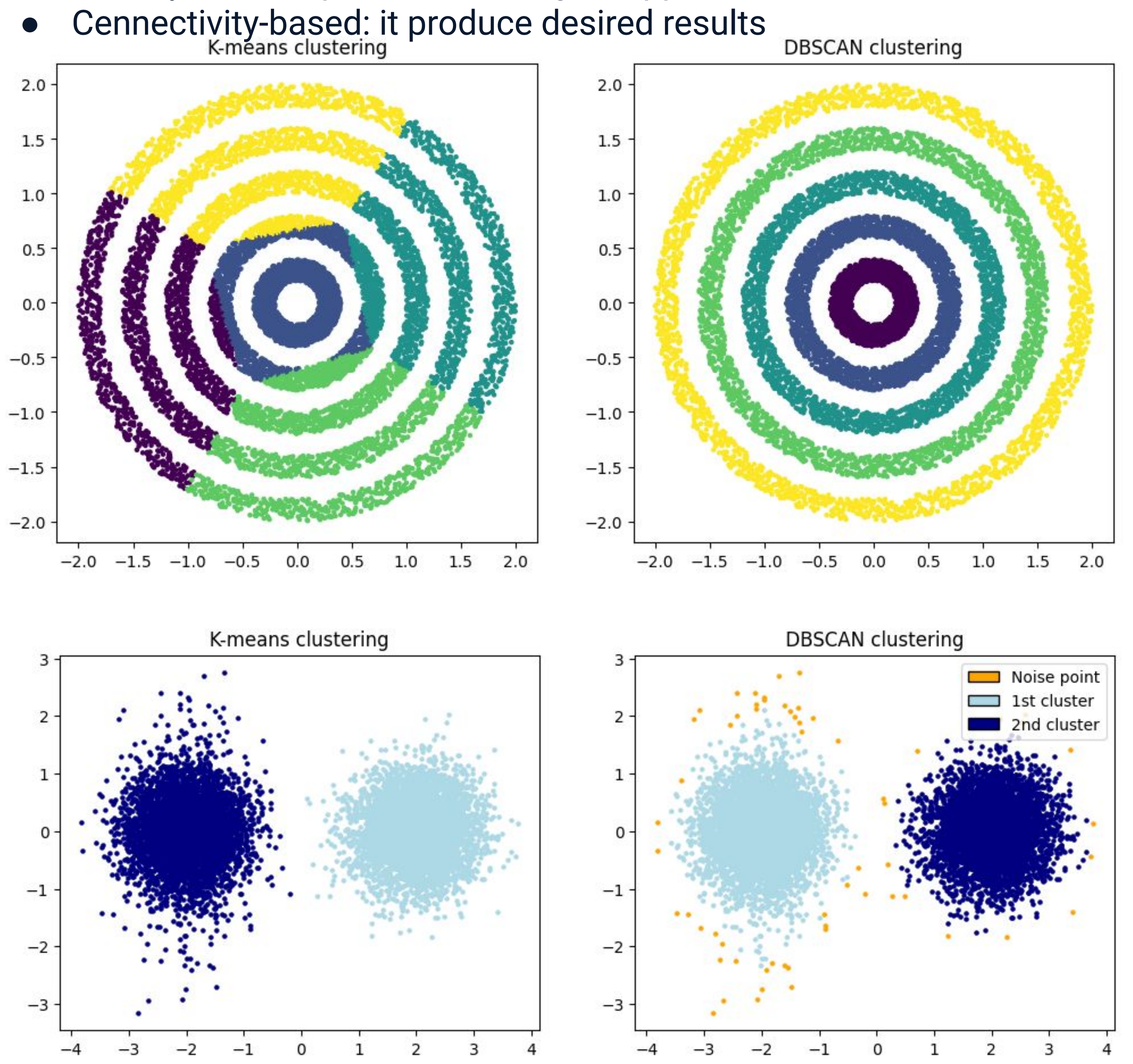
From Python to C++: A Journey to Efficient DBSCAN

Student: Wei-Lun Chiu, Jhao-Ting Chen
Carnegie Mellon University, Pittsburgh, PA

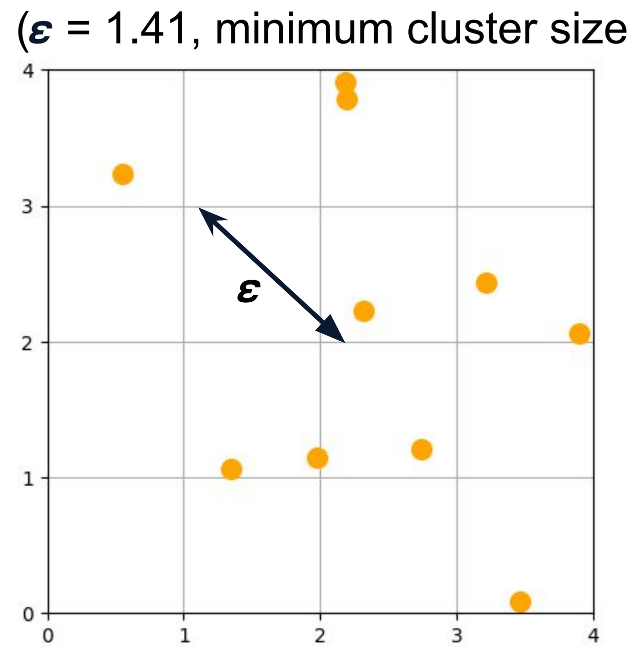
DBSCAN: What, Why, How?

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

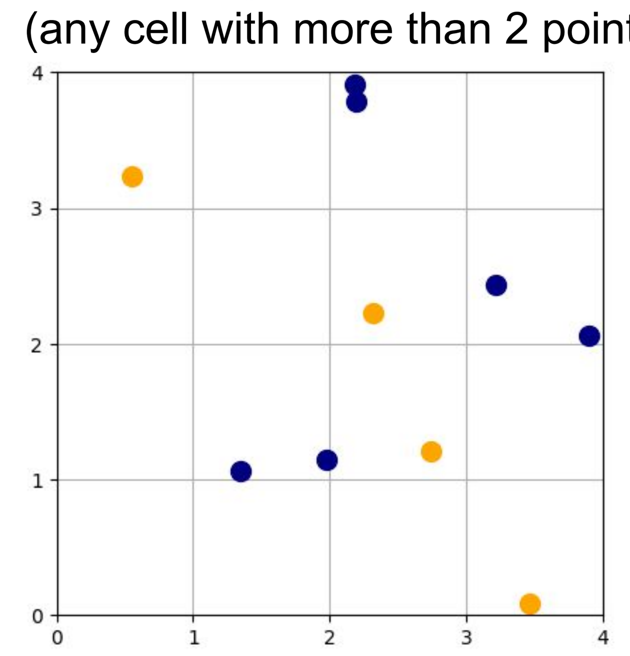


- Grid-based DBSCAN Visualization (ε is minimum connectivity threshold)

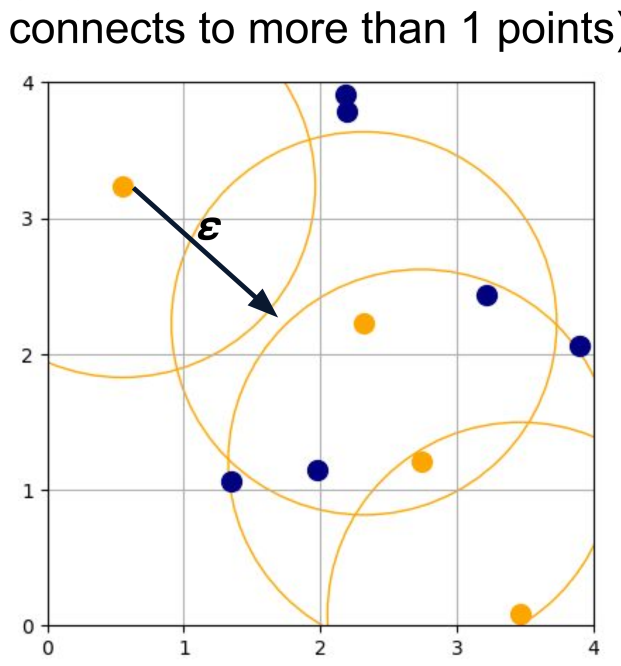
1. Initial Points
(ε = 1.41, minimum cluster size = 2)



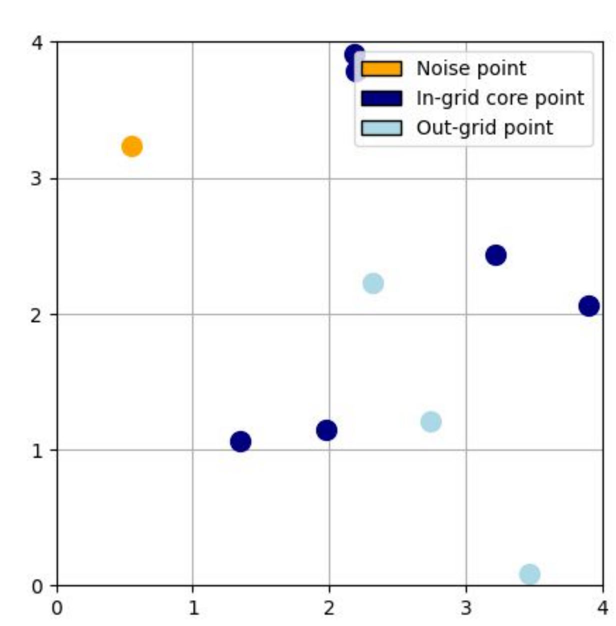
2. Mark **In-grid Core Cells**
(any cell with more than 2 points)



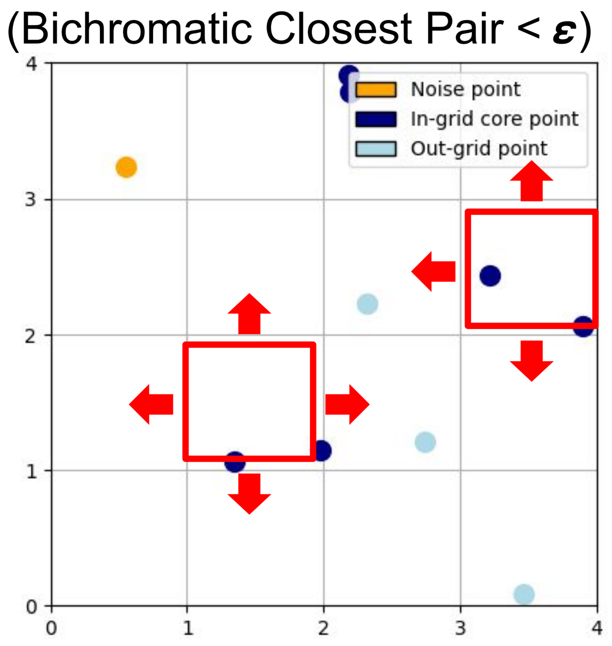
3. Mark **Out-grid Core Cells**
(any non-core cell contain a point connects to more than 1 points)



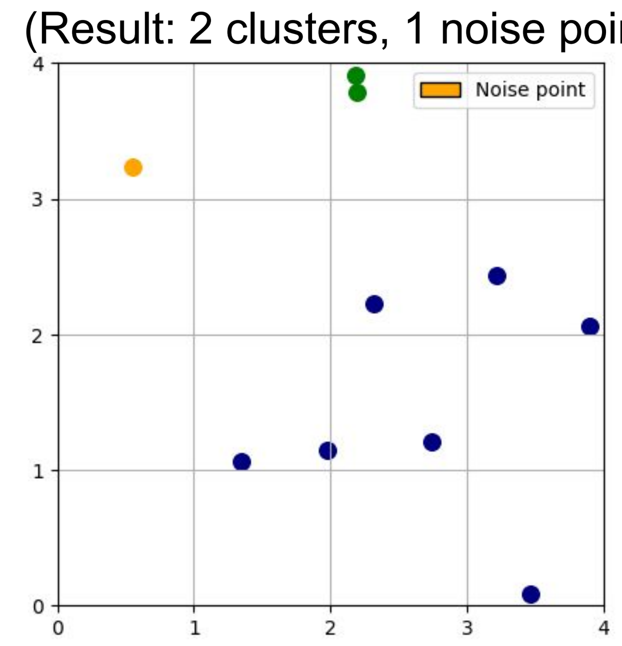
4. Finalized Point Classification
(core point and noise point)



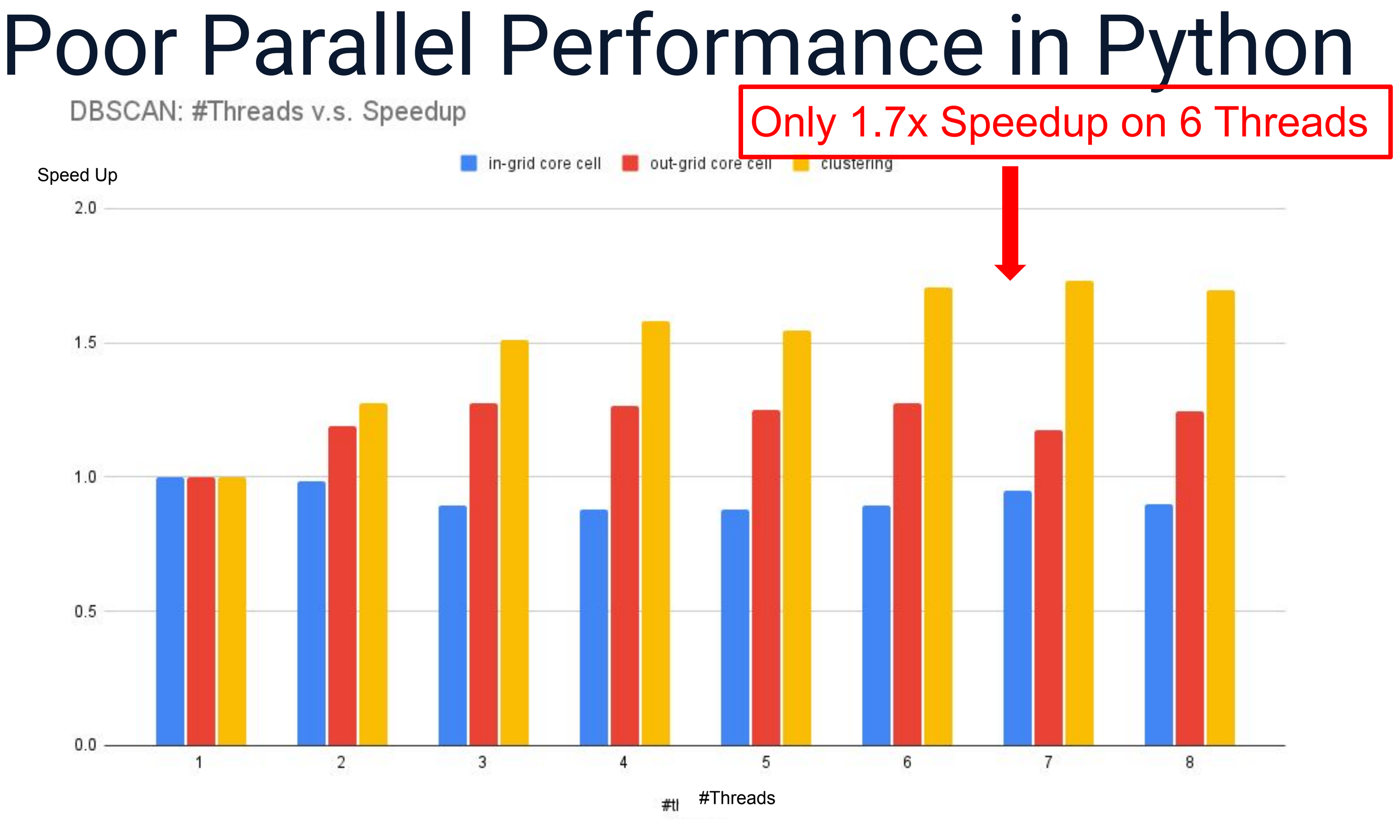
5. Core Cell Clustering Expansion
(Bichromatic Closest Pair < ε)



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%**

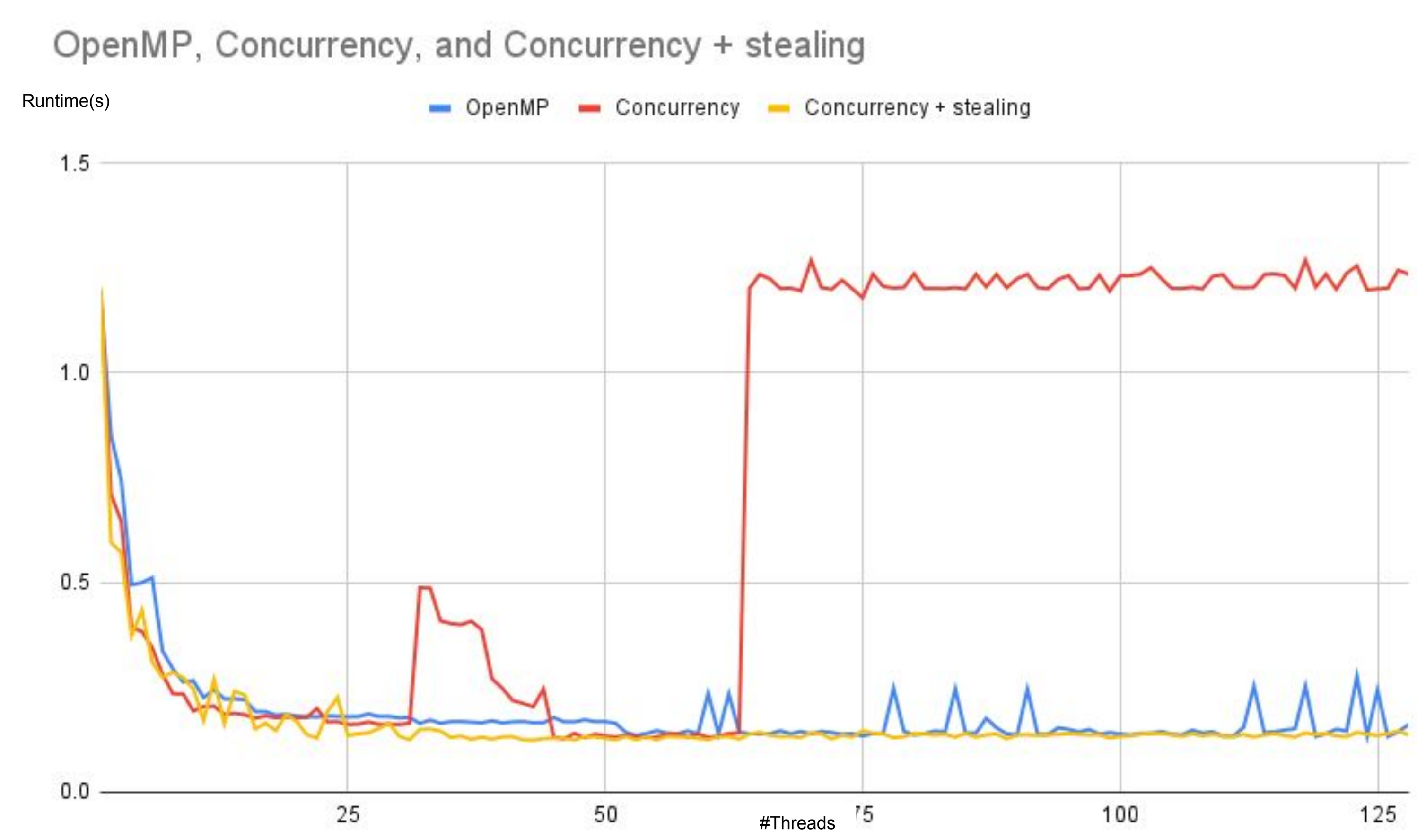
Table 1: Runtime and percentage breakdown for each step of the procedure

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`.

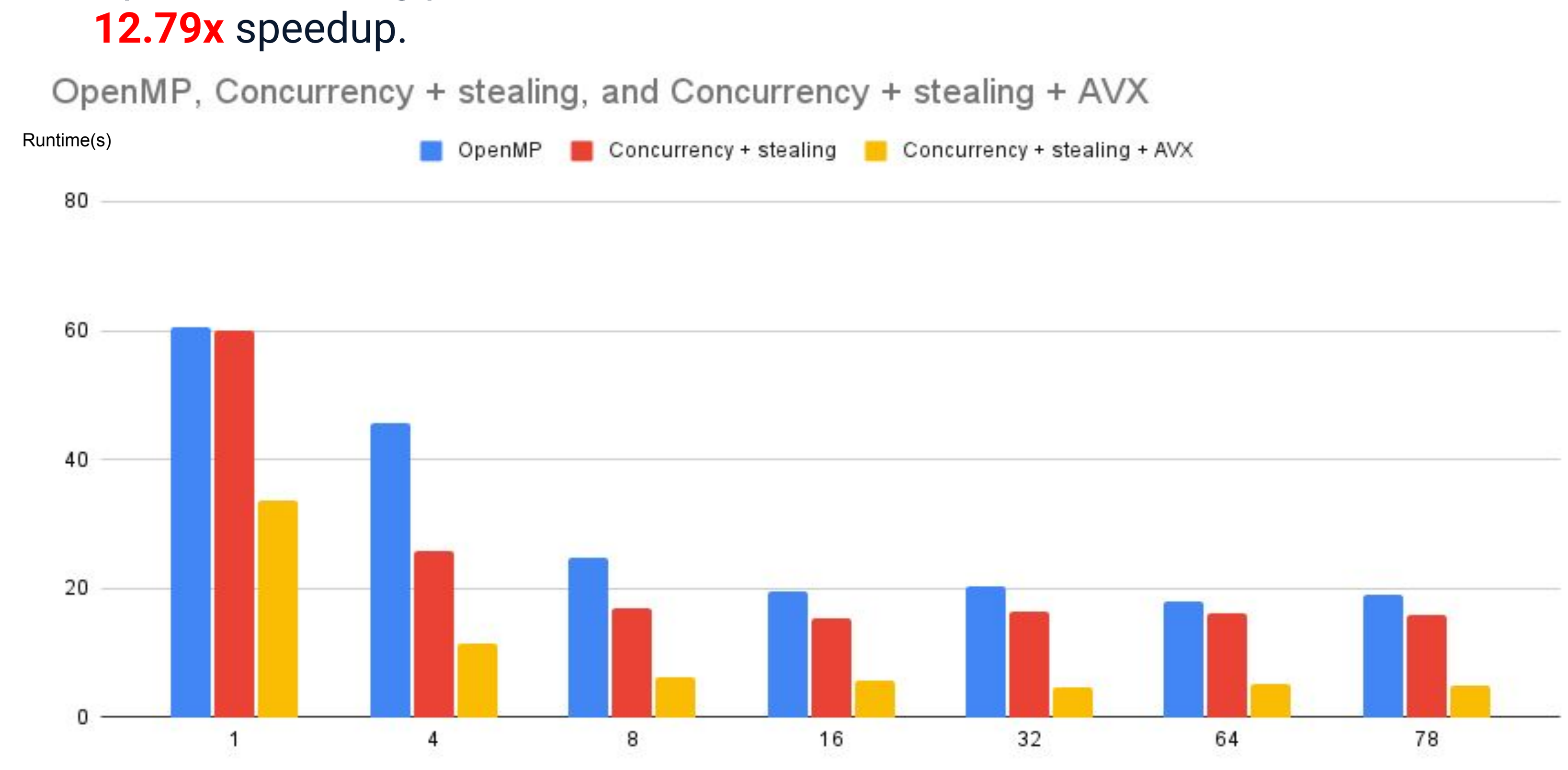


Performance Analysis with Perf

- 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