Parallel Apriori
Algorithm Performance Evaluation

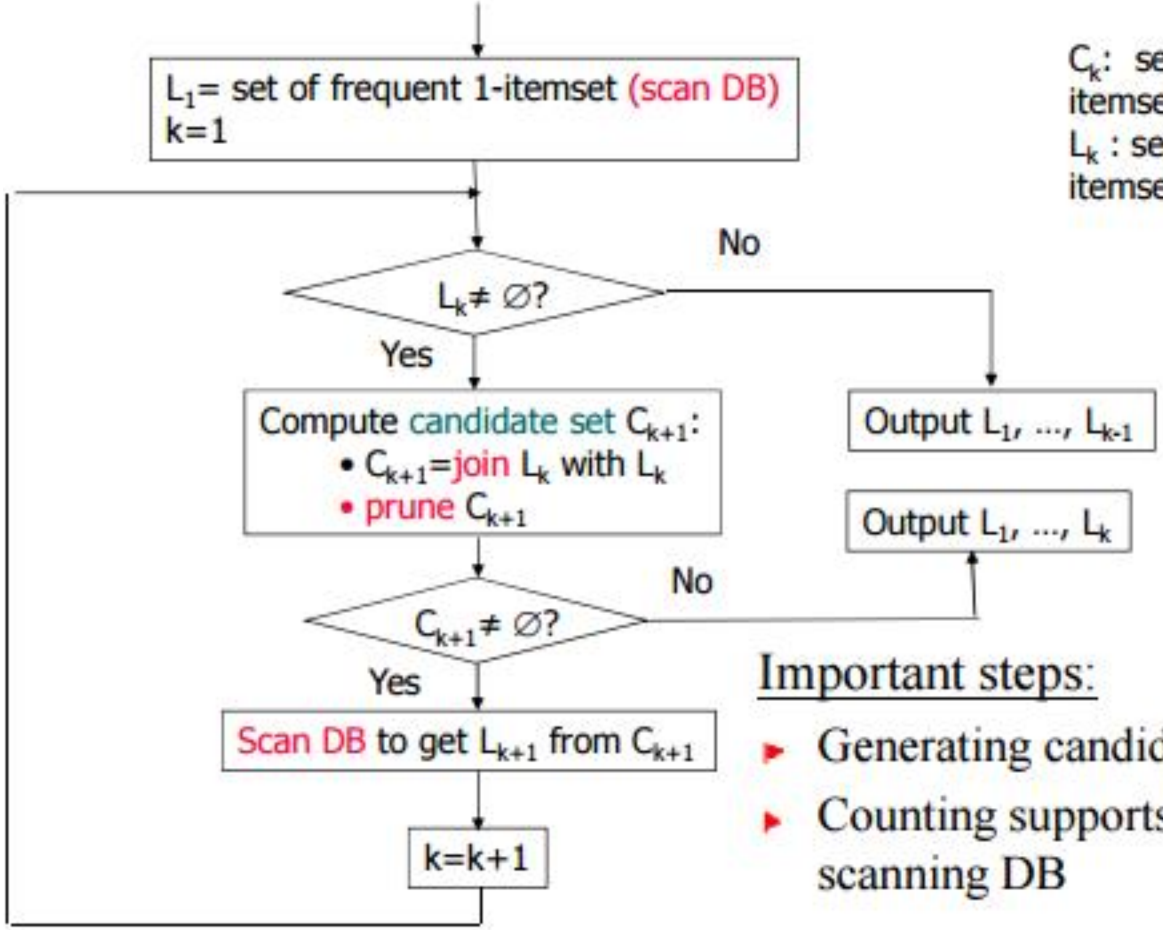
Vassil Halatchev

Department of Electrical Engineering and Computer Science

York University, Toronto

December 1, 2015

Apriori Algorithm (Flow Chart)



C_k: set of candidate k-itemsets
L_k : set of frequent k-itemsets

Important steps:

- Generating candidates
- Counting supports of candidates by scanning DB

Parallel Apriori Algorithms

- **Count Distribution –** each thread generates same candidates at each pass as every other thread
- **Count Distribution Static Map** (new) – same as **CD** but threads update support counts concurrently
- **Data Distribution** – every node in system must process every database transaction

Experimental Setup: Parameters

KEY:

- **Minimum Support**
- **Number of Threads**
- **Number of CPU cores** (i.e. taskset)
- **D :** number of Transactions
- **T :** average number of items per transaction
- **N :** number of different items in the dataset
- **I :** average length of frequent itemset/maximal pattern

EXTRA (fixed):

- **P:** number of patterns (fixed/default: 10000)
- **C:** correlation between patterns(fixed/default: 0.25)
- **R:** average confidence in a rule(fixed/default: 0.75)

Experimental Setup: Tools

- Used **IBM's Quest Synthetic Data Generator** (this is the Benchmark tool) for association rule mining
- Used the login node at Manycore Testing Lab to submit the experimental tasks

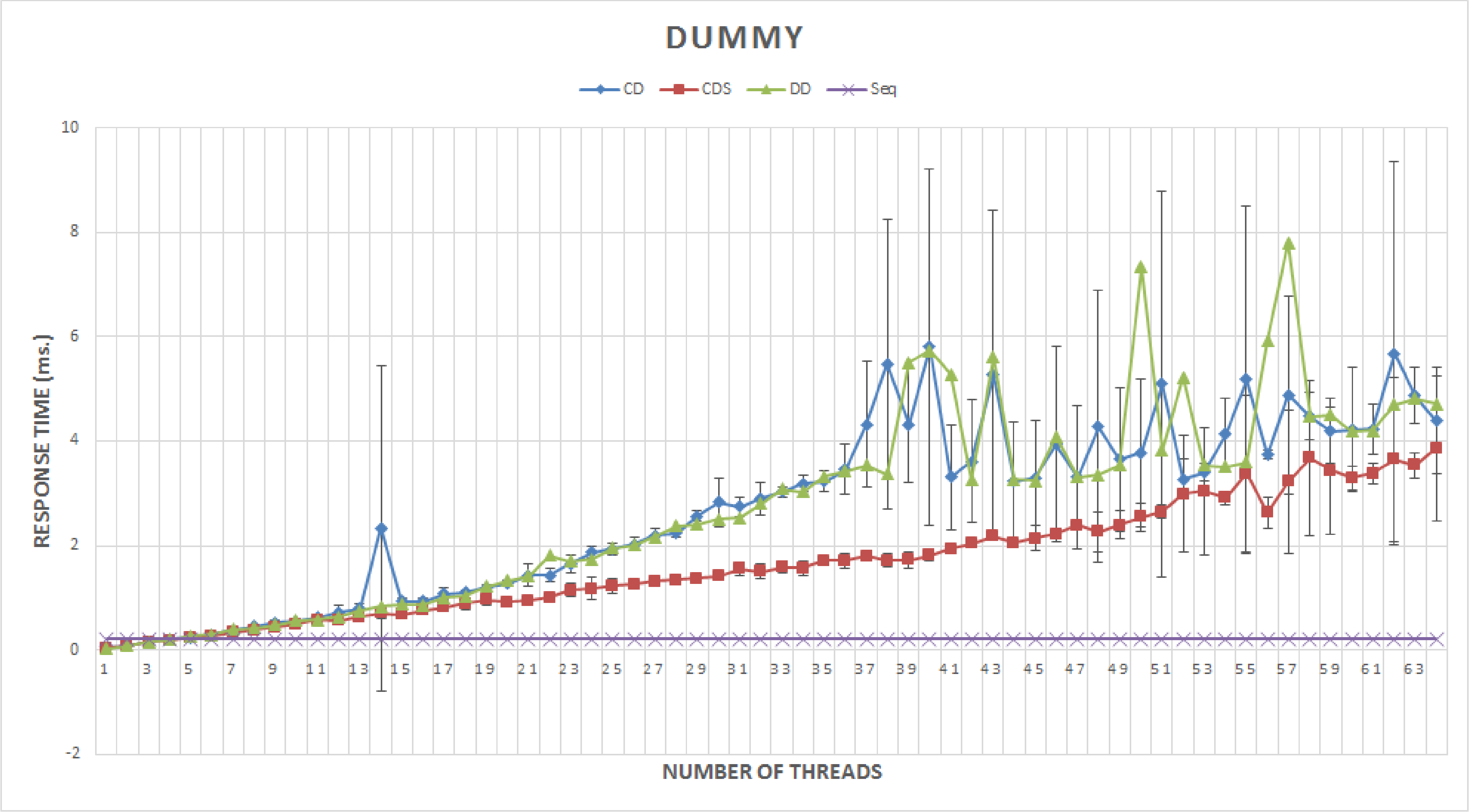
Experimental Setup: Test Measurement

- **Response time was measured** as the time elapsed from the initiation of the execution of the first thread to the end time of the last thread finishing the computation
- **Apache Commons Math 3.5** was used to calculate the **mean** and **standard deviation**
- **Each configuration** was ran **10 times,** ignoring results from **first 4 runs**
- **-server** and **–d64** passed as arguments to **JVM**

Performance Experiment 1 (Dummy)

- **All 4** Implementations (CD, CDS, DD and Sequential) were tested on configuration:
  - **Minimum Support :** irrelevant
  - **Number of Threads :** {1..64} (Note: Sequential was ran on a single thread)
  - **CPU's :** 32
  - **Empty Database**

Experiment 1:DUMMY

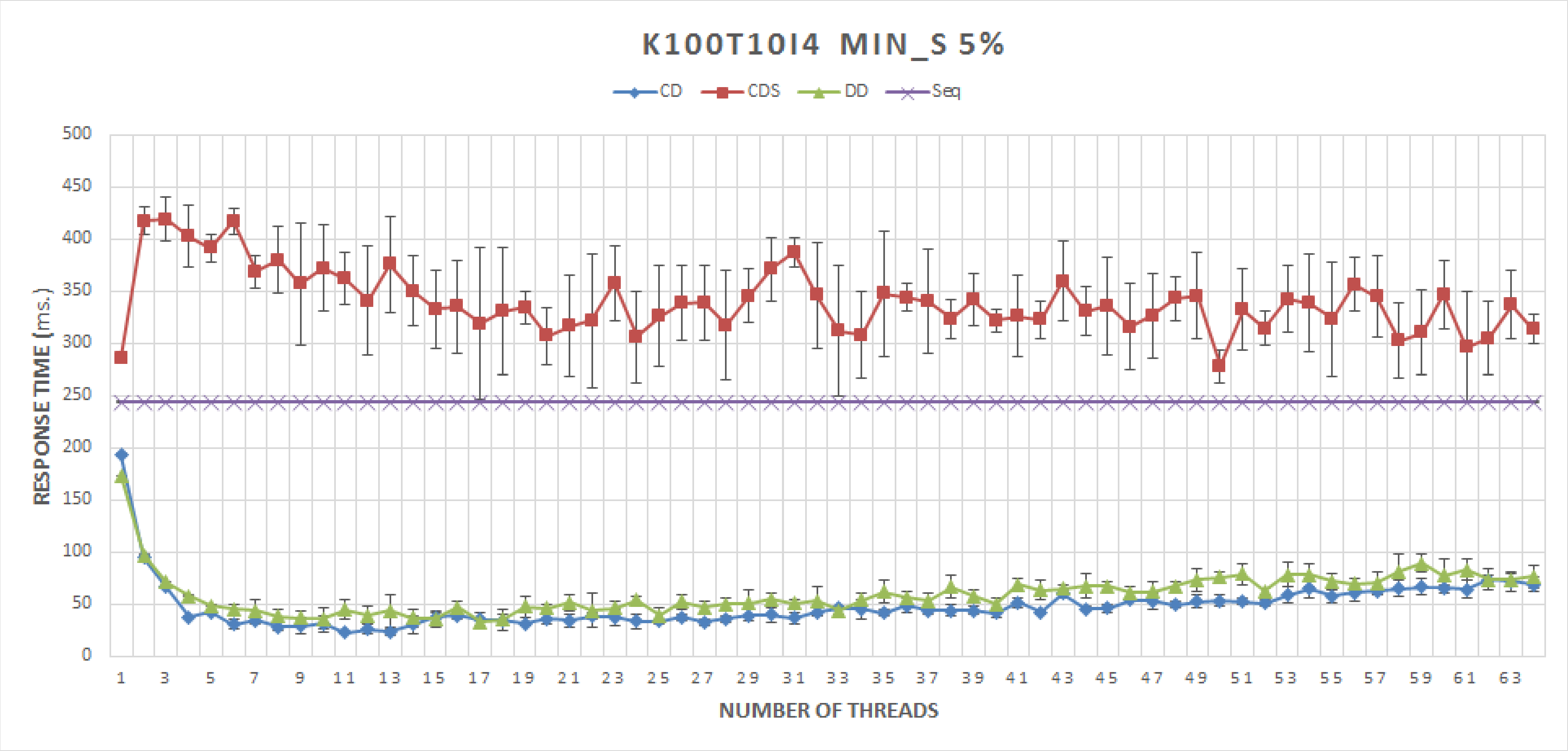


Performance Experiment 2

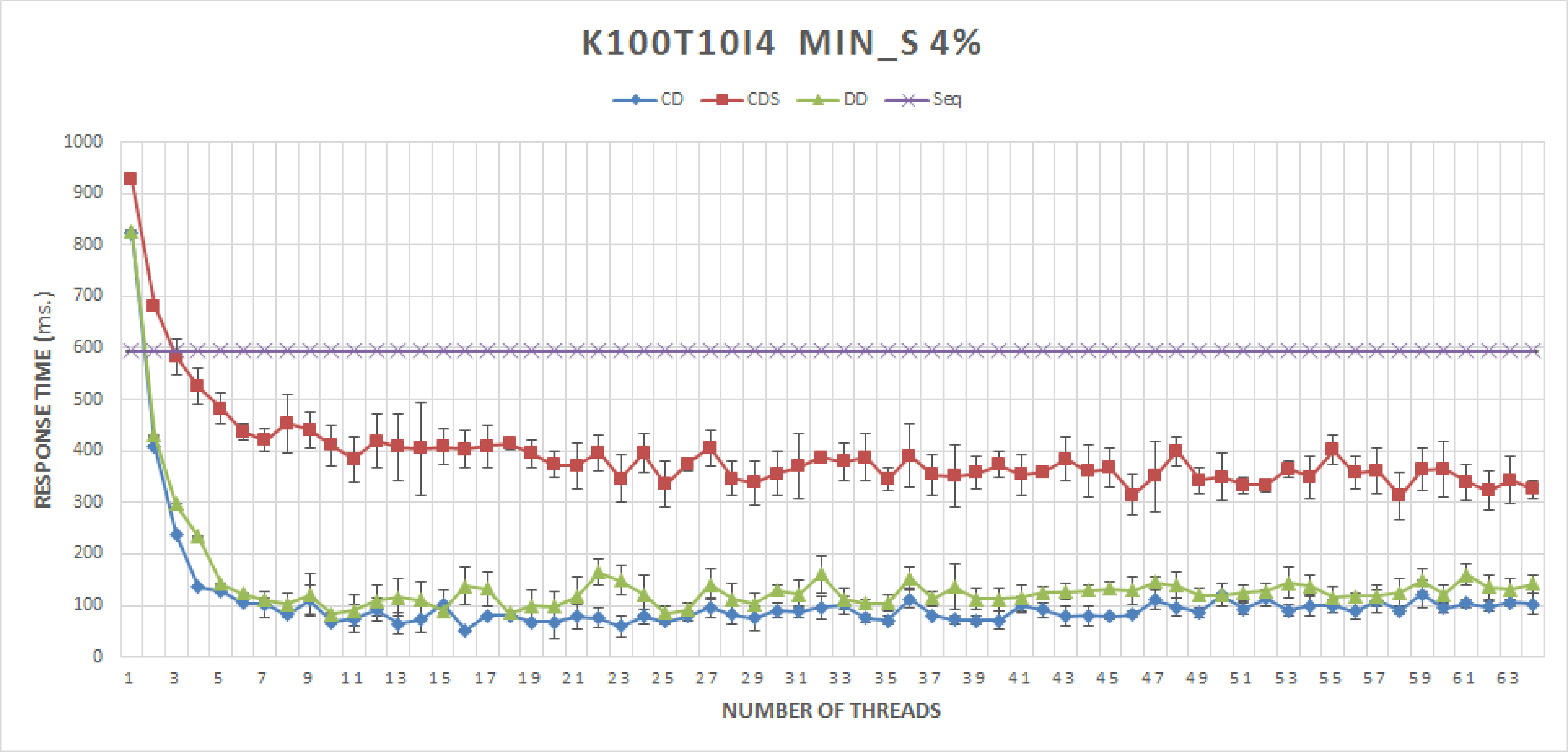
- **All 4** Implementations (CD, CDS, DD and Sequential) were tested on configuration:
  - **Minimum Support :** 0.05 (i.e. 5%), 0.04, 0.03, 0.02, 0.01, 0.005
  - **Number of Threads :** {1..64} (Note: Sequential was ran on a single thread)
  - **CPU cores :** 32
  - **K :** 100(in 000's)
  - **T :** 10
  - **I :** 4
  - **N :** 1(in 000's)

YORK UNIVERSITÉ UNIVERSITY

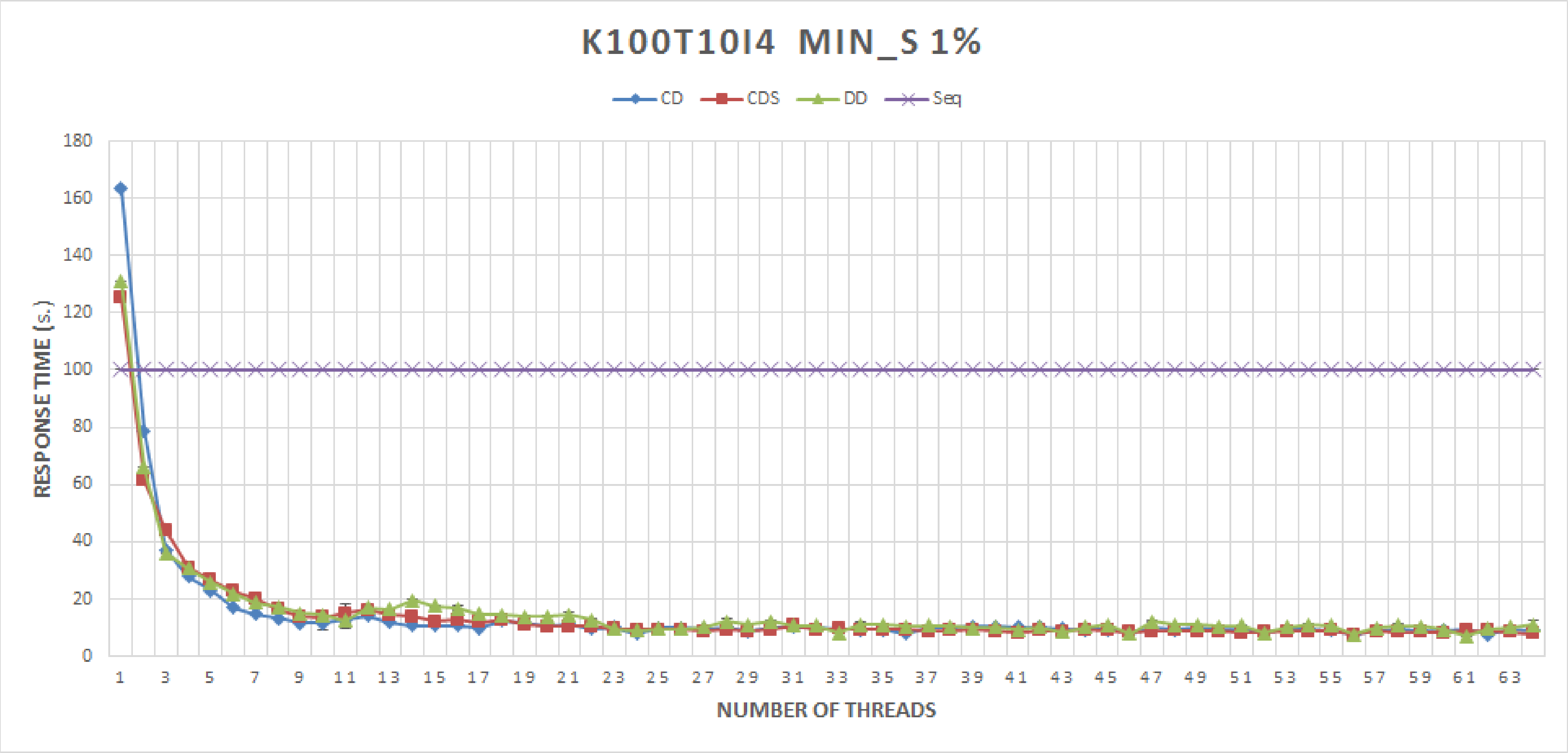
Experiment 2: Minimum Support 5%



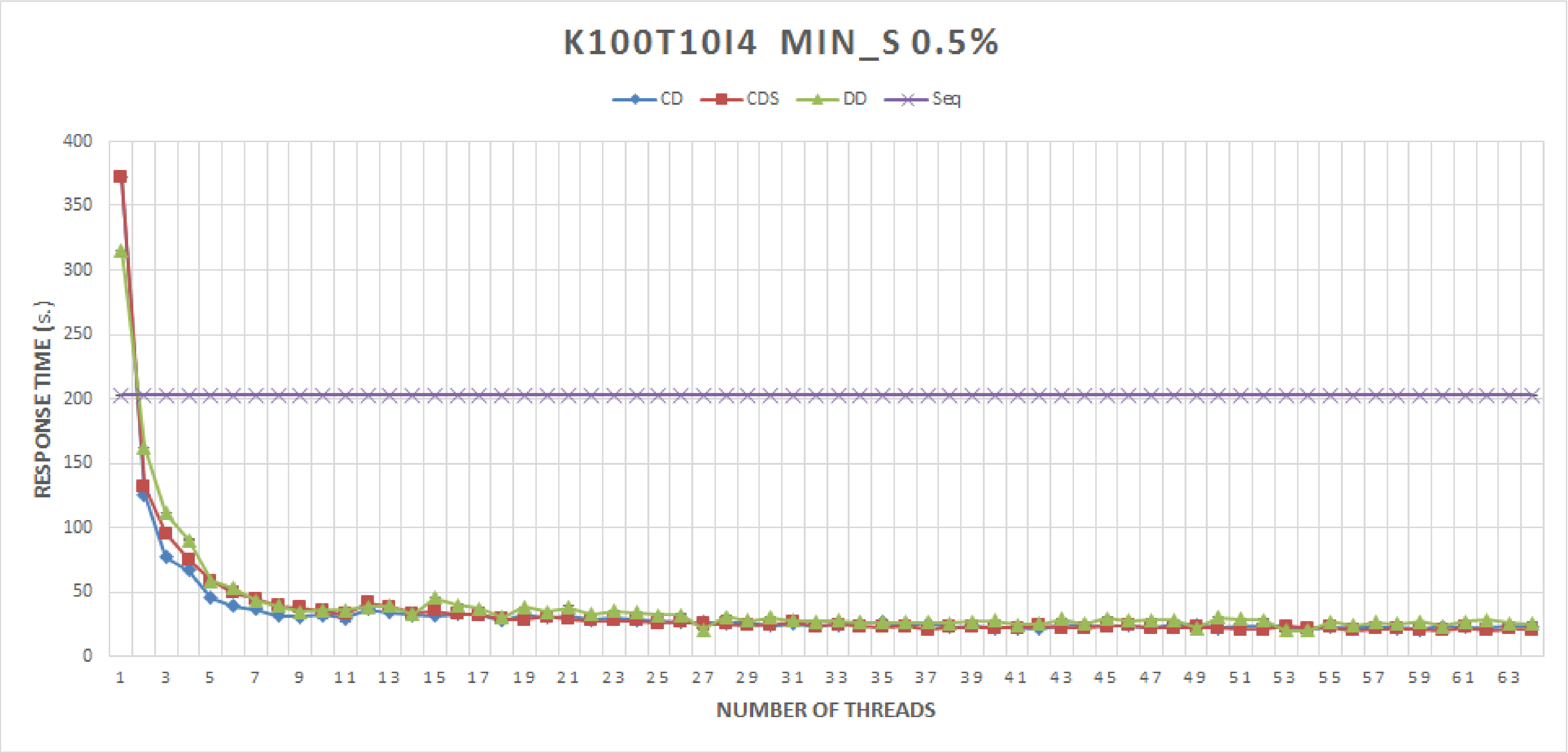
Experiment 2: Minimum Support 4%



Experiment 2: Minimum Support 1%



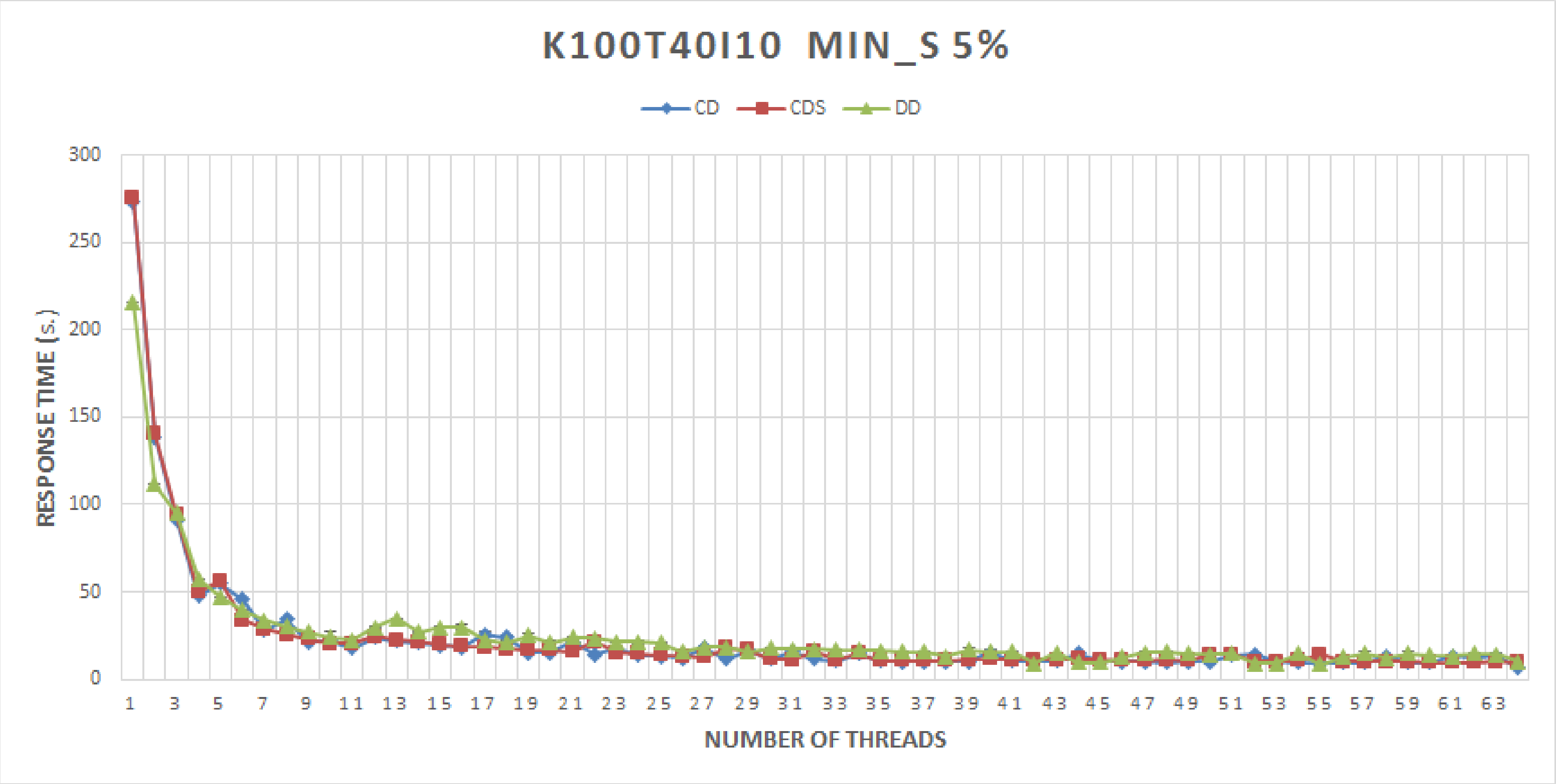
Experiment 2: Minimum Support 0.5%



Performance Experiment 3

- **All 4** Implementations (CD, CDS, DD) were tested on configuration:
  - **Minimum Support :** 0.05 (i.e. 5%)
  - **Number of Threads :** {1..64}
  - **CPU cores :** 32
  - **K :** 100 (in 000’s)
  - **T :** 40
  - **I :** 10
  - **N :** 1 (in 000’s)

Experiment 3: Minimum Support 5%

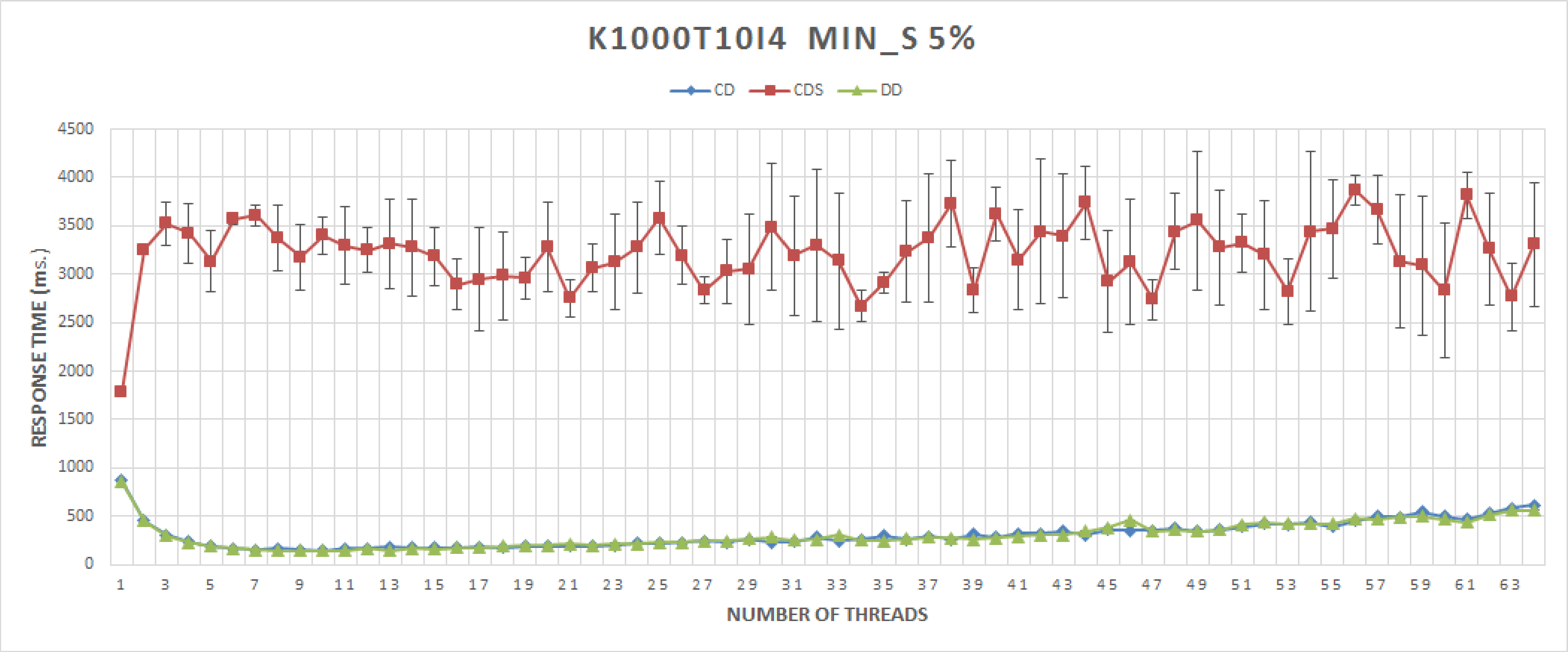


Performance Experiment 4

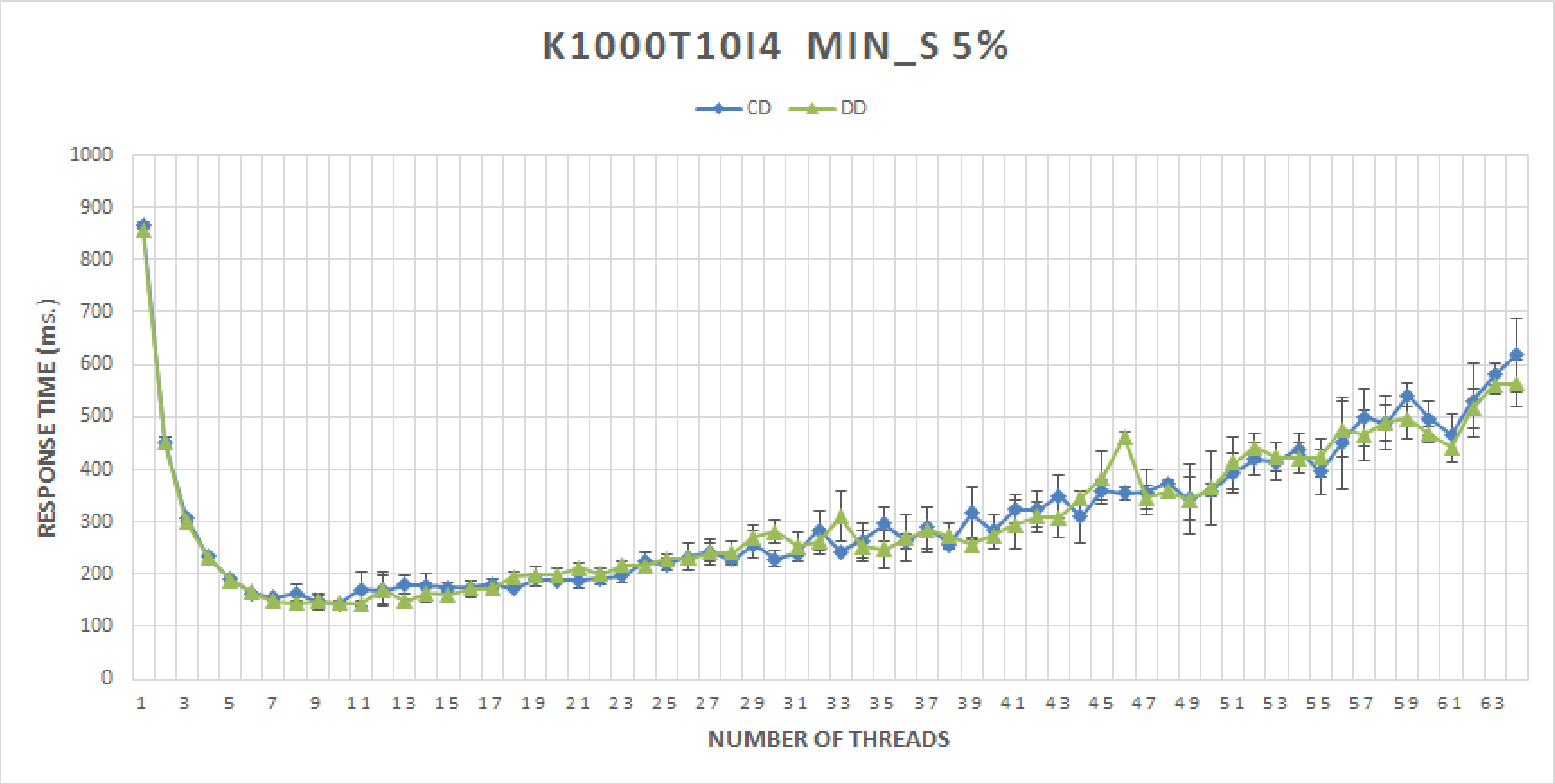
- **All 4** Implementations (CD, CDS, DD) were tested on configuration:
  - **Minimum Support :** 0.05 (i.e. 5%), 0.04
  - **Number of Threads :** {1..64}
  - **CPU cores :** 32
  - **D :** 1000 (in 000's)
  - **T :** 10
  - **I :** 4
  - **N :** 10 (in 000's)

YORK UNIVERSITÉ UNIVERSITY

Experiment 4: Minimum Support **5%**

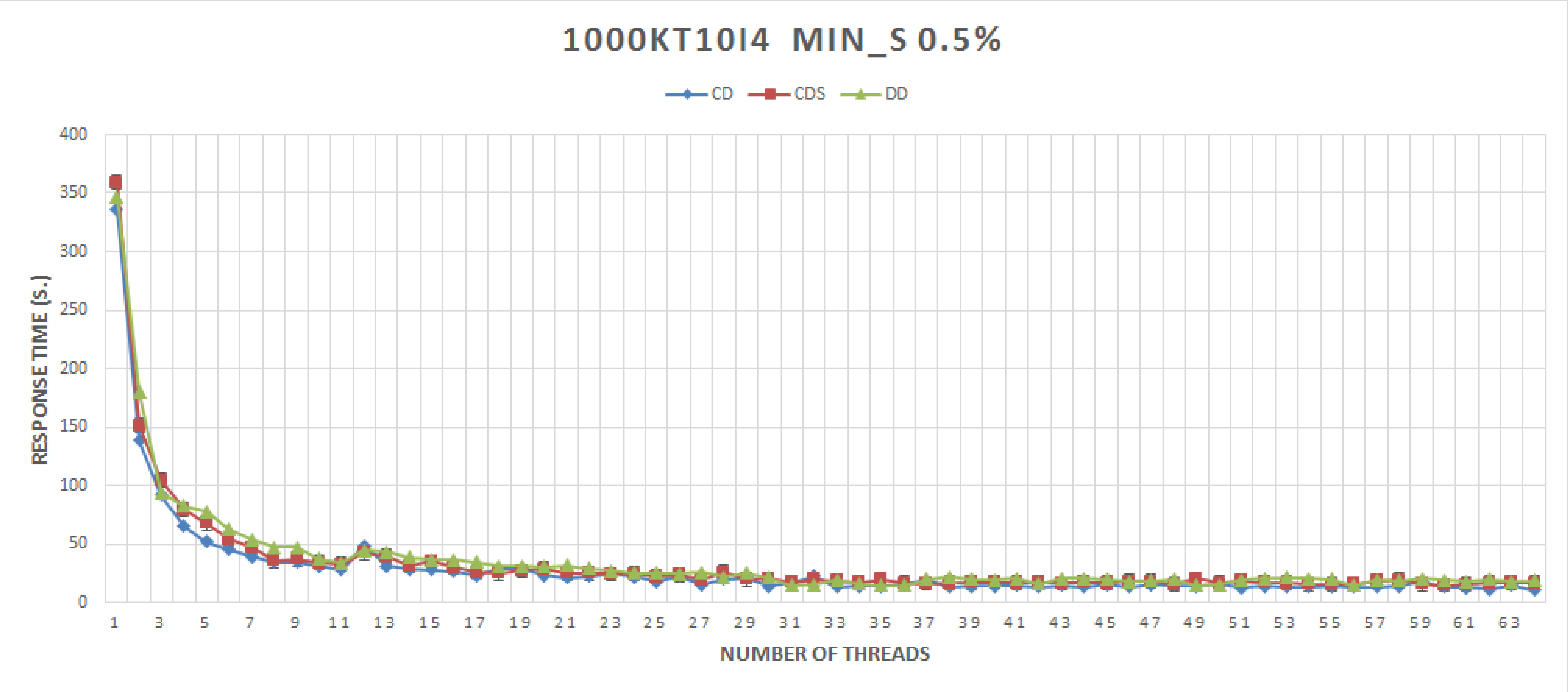


Experiment 4: Minimum Support **5%** looking only at **CD** & **DD**



Thread # 1- 6: database division is beneficial
Thread #11-end: database division is detrimental

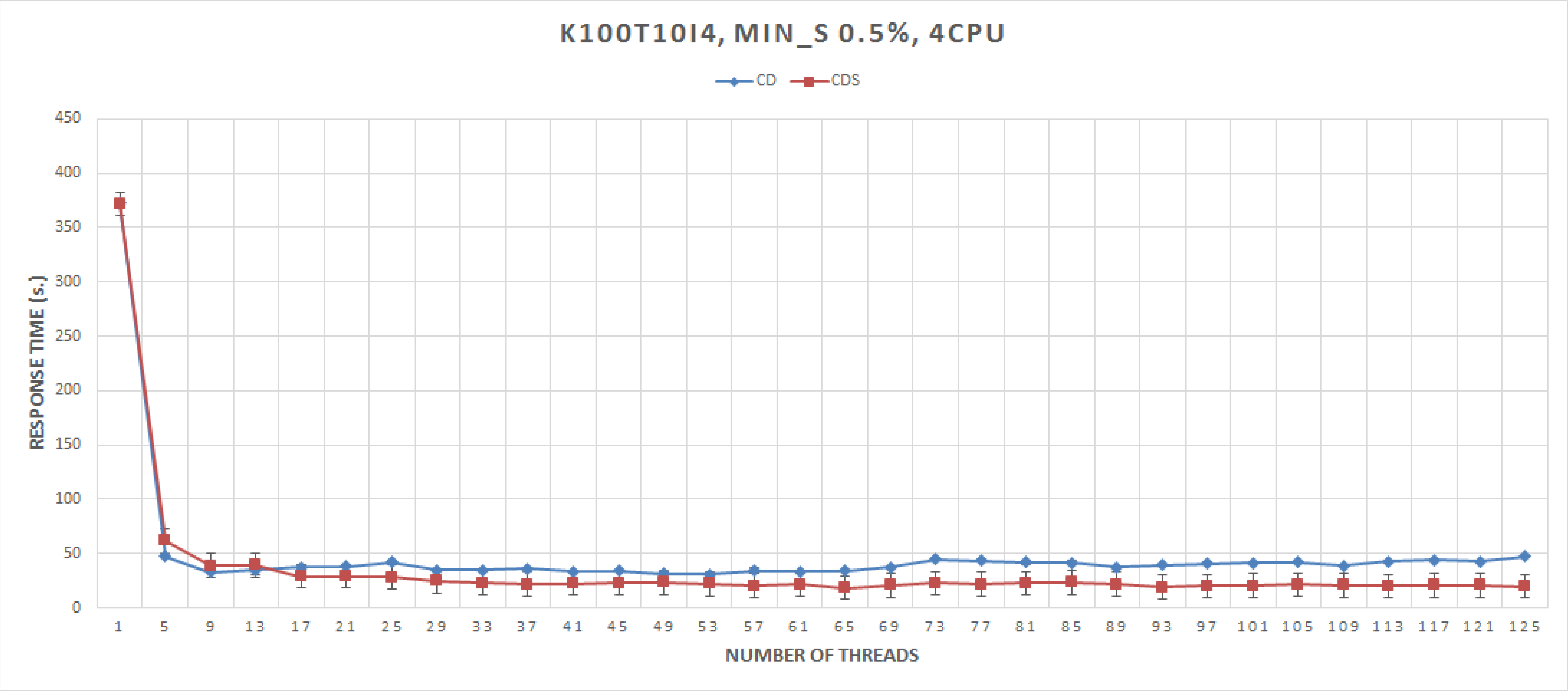
Experiment 4: Minimum Support **0.5%**



Performance Experiment 5

- **All 4** Implementations (CD, CDS) were tested on configuration:
  - **Minimum Support :** 0.05 (i.e. 5%)
  - **Number of Threads :** {1..128}
  - **CPU cores :** 4
  - **K :** 100 (in 000's)
  - **T :** 40
  - **I :** 10
  - **N :** 1 (in 000's)

Experiment 4: Minimum Support **0.5%** on **4 cores**



Hypothesis & Reasoning

1. **Small database** (e.g. 100K) and **a large number of candidates per pass**

you would expect **DD > CD & CDS**

- **Reasoning: DD** will go through **candidates per pass faster,** as they are distributed among threads, than **CD** and **CDS**, and the database is small so it won't hinder its performance there

2. **Large database** (e.g. 10000K+) and **a small number of candidates per pass**

you would expect **CD & CDS > DD**

- **Reasoning:** The running time cost of having to parse the whole database by each thread at every pass will be greater than the cost incurred by **CD** or **CDS**

Further Testing

- **ScaleUp**(Increase the number of threads and database size proportionally)

e.g. Number of threads = 1, database size = 1GB

Number of threads = 64, database size = 64GB

while keeping the result constant (i.e. number of candidates whose support must be summed remains constant**)**

- **Relative ScaleUp** (number of threads = 1, database size = 1GB will be our reference point)
- **SizeUp :** fix the number of threads (e.g. 32) and increase the size of the database each node holds

of Candidates (for K100T10I4)

Minimum support: 0.05
Number of Candidates generated: 55
Number of Frequent itemsets found: 10
Number of levels: 1
Minimum support: 0.04
Number of Candidates generated: 351
Number of Frequent itemsets found: 26
Number of levels: 1
Minimum support: 0.03
Number of Candidates generated: 1830
Number of Frequent itemsets found: 60
Number of levels: 1
Minimum support: 0.02
Number of Candidates generated: 12090
Number of Frequent itemsets found: 155
Number of levels: 1
Minimum support: 0.01
Number of Candidates generated: 70501
Number of Frequent itemsets found: 385
Number of levels: 3
Minimum support: 0.005
Number of Candidates generated: 162389
Number of Frequent itemsets found: 1073
Number of levels: 5