Concurrent K-Means Algorithm

Verification

COSC 6490A
Miroslaw Kuc
York University May 14, 2009

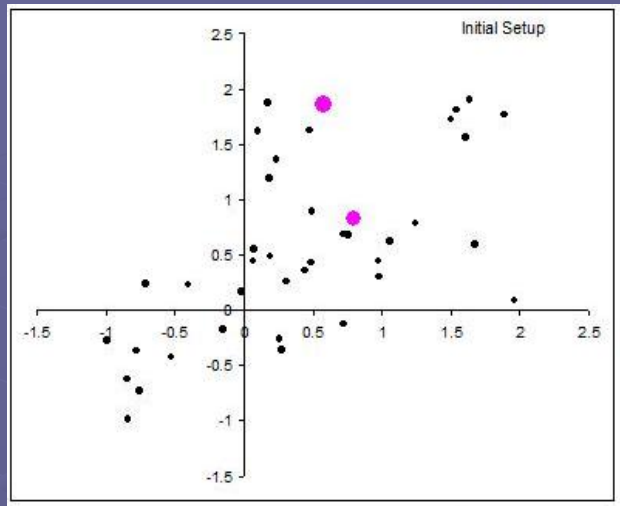
K-Means Algorithm

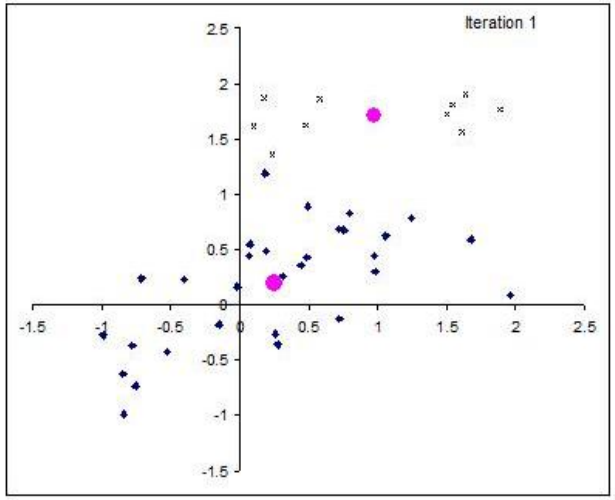
One of the most popular clustering algorithms.

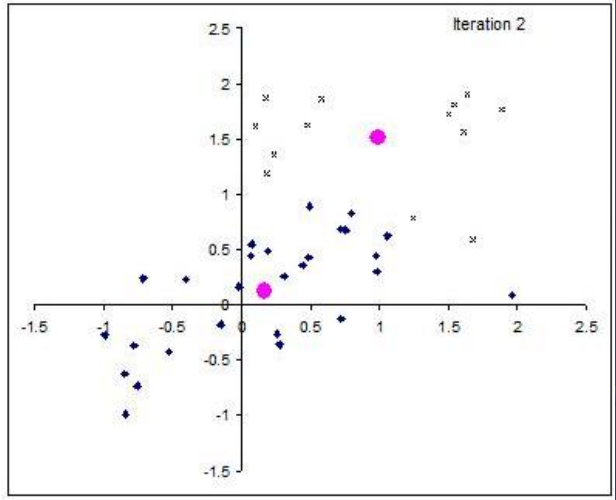
1. Randomly assign cluster centers (e.g. select random points from within the dataset)
2. For each point calculate the distance to the cluster centers and assign the point to the closest “cluster”.
3. Based on the membership calculated in step (2) calculate the center of the new clusters.
4. Repeat steps (2) and (3) until stopping criteria are met (e.g. no point change cluster membership).

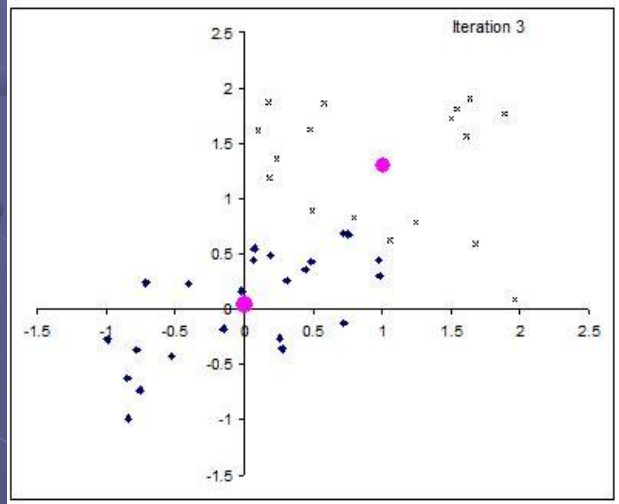
MacQueen, 1967 [1]

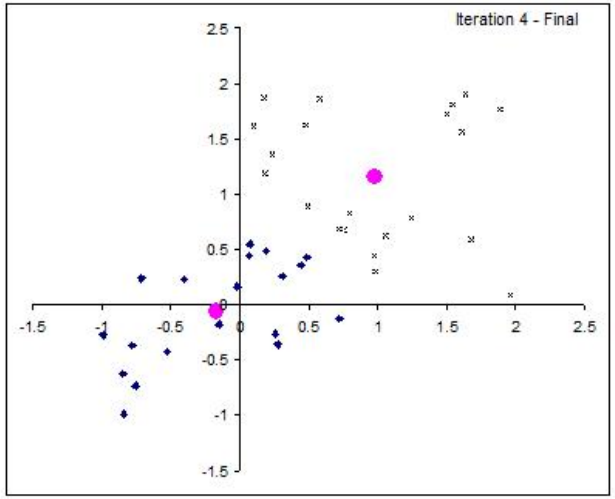
K-Means Algorithm











Brute Force: 549,755,813,800 = $5.5*10^{11}$ possible 2-cluster arrangements

K-Means Algorithm

Project:

Implement single-threaded and 2 methods for “parallelization”

1) Single Threaded Version

2) Distributed Memory Approach

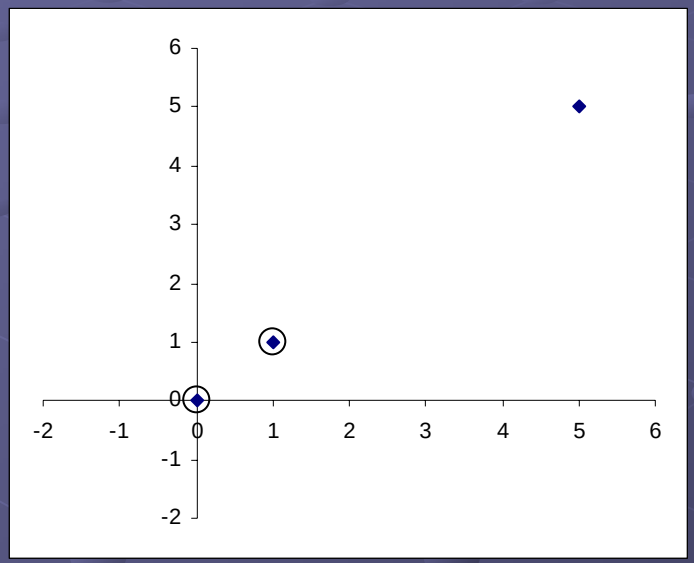
3) Shared Memory Approach

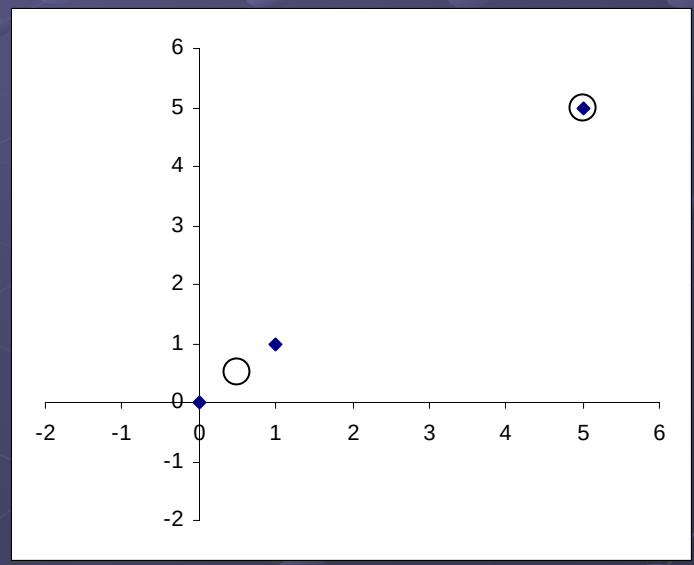
Test Dataset

Could not test the system on the full 100,000 point dataset

(the application runs out of memory with just 8 points)

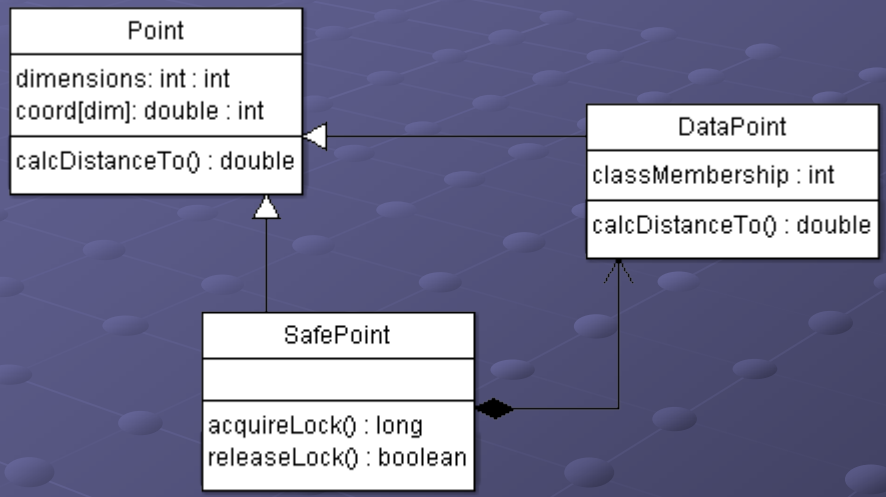
test dataset: 3 points (running time: apx. 5min)





Forces the algorithm to do something

Safe Point



- Stores coordinates of the points
- Allows calculating distance to other points
- Acquiring and Releasing of a lock on data points

Data Locking

```
P(mutex_locked);
if (!locked and !processed) {
        locked = true;
        V(mutex_locked);

        // calculate cluster membership

        P(mutex_membership);
        // set membership
        V(mutex_membership);

        P(mutex_locked);
        processed = true;
        locked = false;
}
V(mutex_locked);
```

Data storage and processing are in different objects.

mutex_membership

Based on the structure of the code determined that the semaphore may not be necessary

```
P(mutex_membership);
// set membership
V(mutex_membership);
```

```
synchronized(this) {
verify_mem_ct++;
}
assert (verify_mem_ct == 1);
// set membership
```

```
====================================================== results
no errors detected

====================================================== statistics
elapsed time:       0:05:37
states:             new=790888, visited=1122977, backtracked=1913864, end=40
search:             maxDepth=403, constraints=0
choice generators:  thread=790887, data=0
heap:               gc=2125038, new=75267, free=155926
instructions:       51253336
max memory:         148MB
loaded code:        classes=101, methods=4816
```

mutex_locked

```
if (!locked and !processed) {
        synchronized(this) {
        this.verify_locked_ct++;
        }
        assert(this.verify_locked == 1);
        locked = true;

        // calculate cluster membership

        // set membership

        processed = true;
        locked = false;
}
```

```
====================================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty "java.lang.AssertionError  at SafePoint.acquireLock..."
```

Data Locking

```
P(mutex_locked);
if (!locked and !processed) {
        locked = true;
        V(mutex_locked);

        // calculate cluster membership

        // set membership

        // P(mutex_locked);
        processed = true;
        locked = false;
} else {
        V(mutex_locked);
}
```

Causes no error in JPF

Order of Operations

```
P(mutex_locked);
if (!locked and !processed) {
        locked = true;
        V(mutex_locked);

        // calculate cluster membership

        // set membership

        locked = false;
        processed = true;
} else {
        V(mutex_locked);
}
```

Causes an error in JPF – order of these two operations is important

Changed Membership

changedMembership flag is used to track if any of the points have changed their cluster membership; thus, if the algorithm is required to run again.

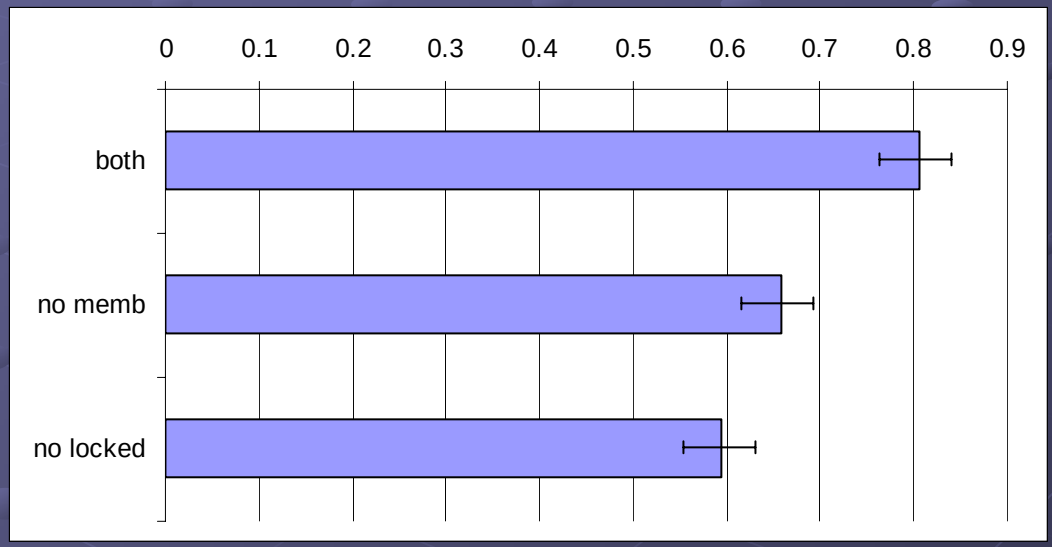
Implemented as a class with a synchronized “write” method.

Checked if the synchronization is indeed necessary.

It is.

Improved Performance

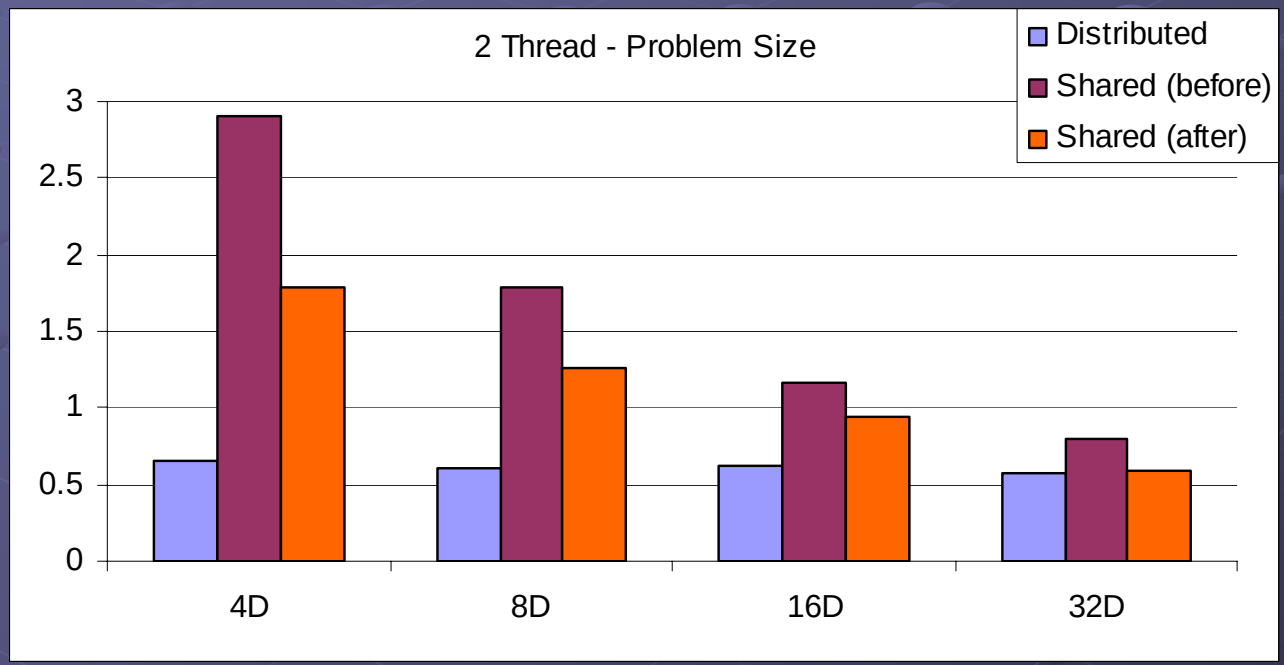
Substantial improvement to the performance when unnecessary locks are removed (fixed problem size: 32D, 2 threads)



26% performance improvement

Improved Performance

Substantial improvement to the performance when unnecessary locks are removed (various problem sizes)



Conclusions

Java appears to have a high cost of thread synchronization; therefore, it is important to examine the necessity of each data lock – may lead to a substantial improvement in performance.

References

[1] MacQueen, J. *Some methods of classification and analysis of multivariate observations*, Proceedings of the fifth Berkeley symposium on mathematical statistic and probability (Vol. 1, pp. 281-297) Berkeley: University of California Press.

Thank You!

Questions?