Concurrent K-Means Algorithm

Implementation

COSC 6490A
Miroslaw Kuc
York University   Apr. 23, 2009

K-Means Algorithm

Project:

Implement single-threaded and 2 methods for “parallelization”

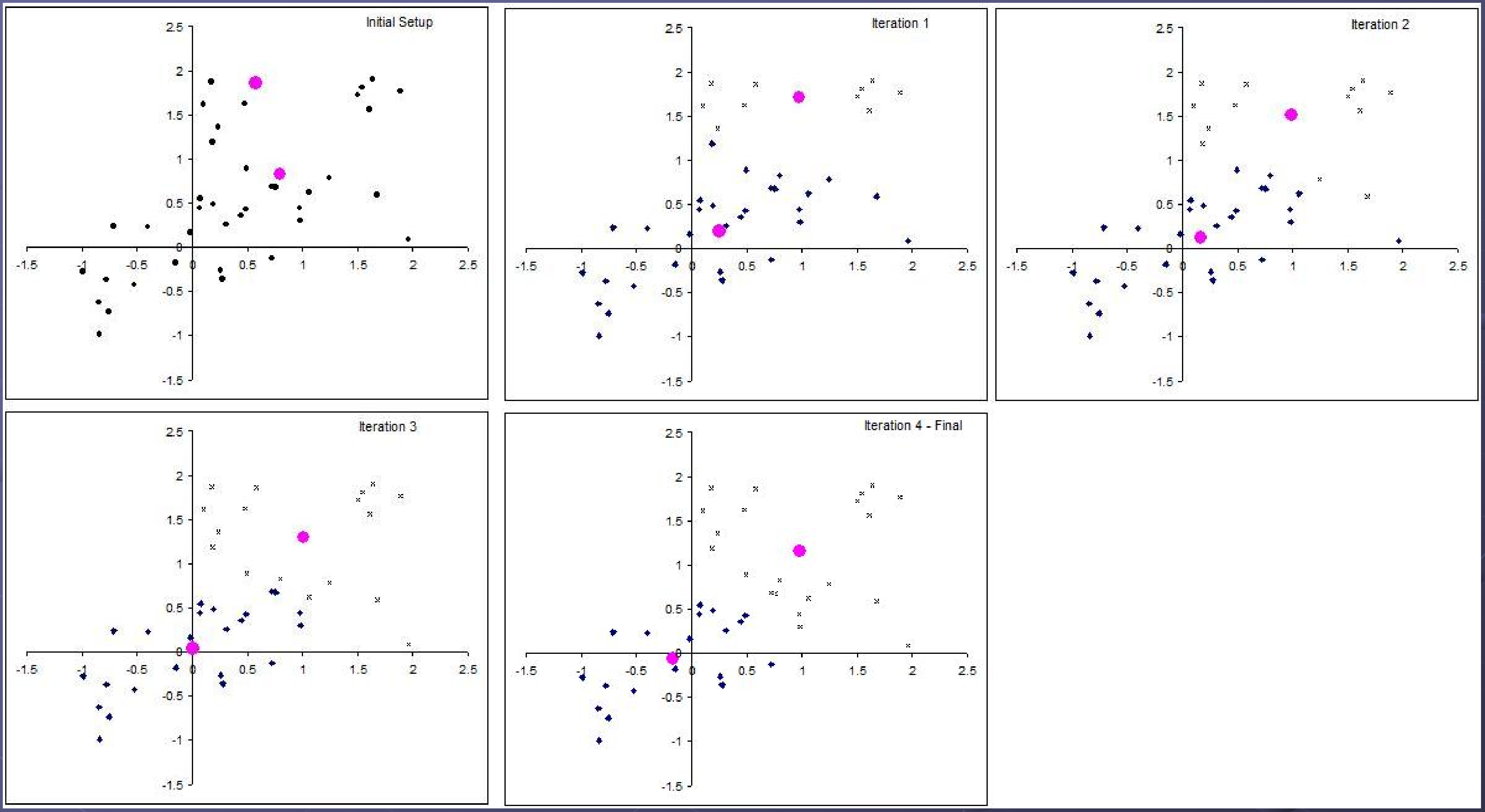
1) Single Threaded Version
2) Distributed Memory Approach
3) Shared Memory Approach

K-Means Algorithm

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

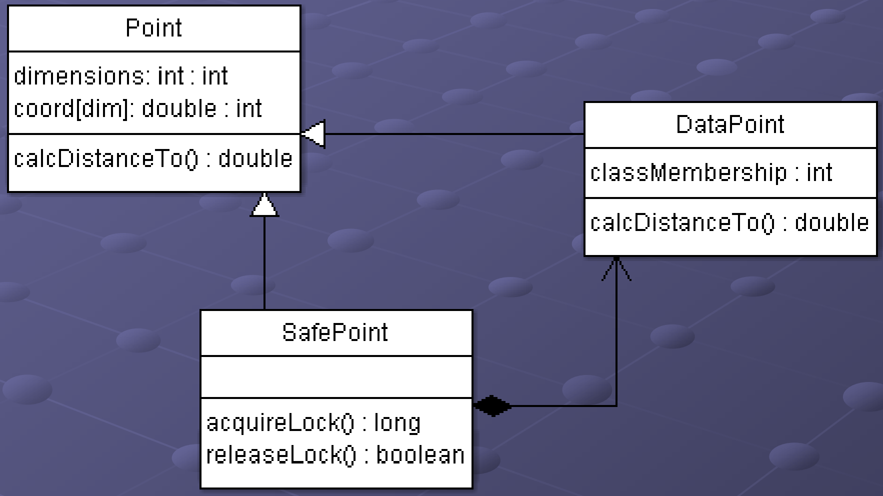
MacQueen, 1967 [1]

k-means Algorithm

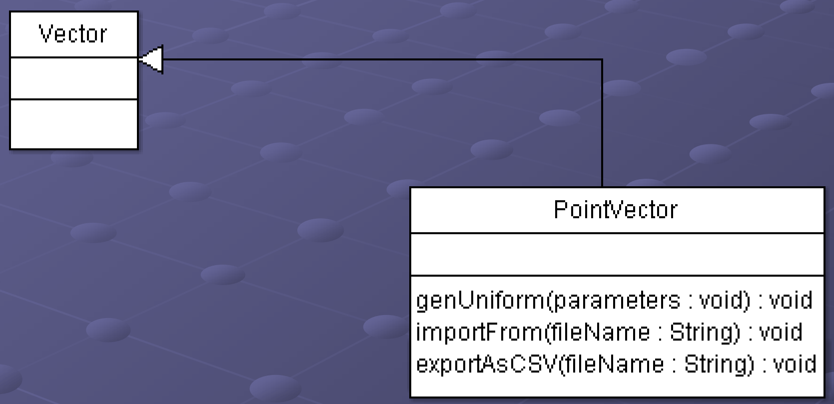


Brute Force: 549,755,813,800 = 5.5*10^{11} possible 2-cluster arrangements

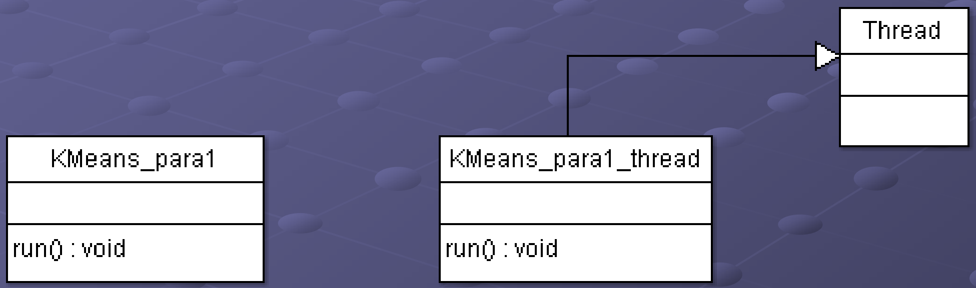
Data Structures



Data Structures



Data Structures



"Distributed" Model Approach

Exploit the "data" parallelism:

- Subdivide the data into equal "chunks": copies points references to a local Vector (otherwise, whole Vector locked)
- Copy cluster centers to a local Vector
- Code very similar to the single-threaded version

```
// for each dataPoint find the nearest cluster center
for (int iPt = 0; iPt < dataPoints.size(); iPt++) {
    int newCluster = -1;
    double newMinDistance = 10E38;
    DataPoint point = (DataPoint) dataPoints.elementAt(iPt);

    for (int iCtr=0; iCtr < clusterCenters.size(); iCtr++) {
        Point center = (Point) clusterCenters.elementAt(iCtr);
        double dist = point.calcDistanceTo(center);
        if (dist < newMinDistance) { // closer center found
            newCluster = iCtr;
            newMinDistance = dist;
        }
    }

    // assign the new cluster center, if it has changed
    if (point.getClusterMembership() != newCluster) {
        point.setClusterMembership(newCluster);
        changedMembership[modThread] = true;
    }
}
```

Data Locking

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

        // calculate cluster membership

        P(mutex_locked);
        V(mutex_membership);
        processed = true;
}
V(mutex_locked);
```

However, data storage and processing are in different objects.

Data Locking

```
// get a lock on the point
public long acquireLock() {
    long dataLock = 0;
    try {
        mutex_locked.acquire();
        if (!locked && !processed) {
            //give the lock only when the point has not been processed
            locked = true;
            mutex_locked.release();
            while (dataLock == 0) { // do not allow a "zero" lock
                dataLock = rand.nextLong();
            }
            this.randDataLock = dataLock;
            mutex_membership.acquire(); //lock up the membership calculation
        } else {
            mutex_locked.release();
        }
        return dataLock;
    } catch (InterruptedException e) {
        System.out.println("SafePoint.aquireLock(): " + e.getMessage());
        return 0;
    }
}
```

Point can only be locked for processing once (flag reset between iterations)
Lock exclusivity accomplished though a randomized “key”.

Data Locking

```
// release the lock on the point
public boolean releaseLock(long dataLock) {
    boolean lock_released = false;
    try {
        mutex_locked.acquire();
        if (randDataLock != 0 && this.randDataLock == dataLock) {
            if (locked) { // release the lock only if it has previously been acquired
                locked = false;
                processed = true;
                lock_released = true;
                randDataLock = 0;
                mutex_membership.release();
            }
        }
        mutex_locked.release();
        return lock_released;
    } catch (InterruptedException e) {
        System.out.println("SafePoint.releaseLock(): " + e.getMessage());
        return false;
    }
}
```

"mutex_membership" forces waiting for the result.
"key" required to release the lock.

Data Locking

```
public boolean setClusterMembership(long dataLock, int newValue) {
    boolean valueSet = false;
    try {
        mutex_locked.acquire();
        if (randDataLock != 0 && this.randDataLock == dataLock) {
            if (locked) { // allow changes to the cluster membership only if locked
                point.setClusterMembership(newValue);
                valueSet = true;
            }
        }
        mutex_locked.release();
        return valueSet;
    } catch (InterruptedException e) {
        System.out.println("SafePoint.setClusterMembership(): " + e.getMessage());
        return false;
    }
}
```

“key” required to change value.

"Shared" Memory Approach

Multiple threads are potentially accessing the same data points; therefore, need to lock points while calculating cluster centers.

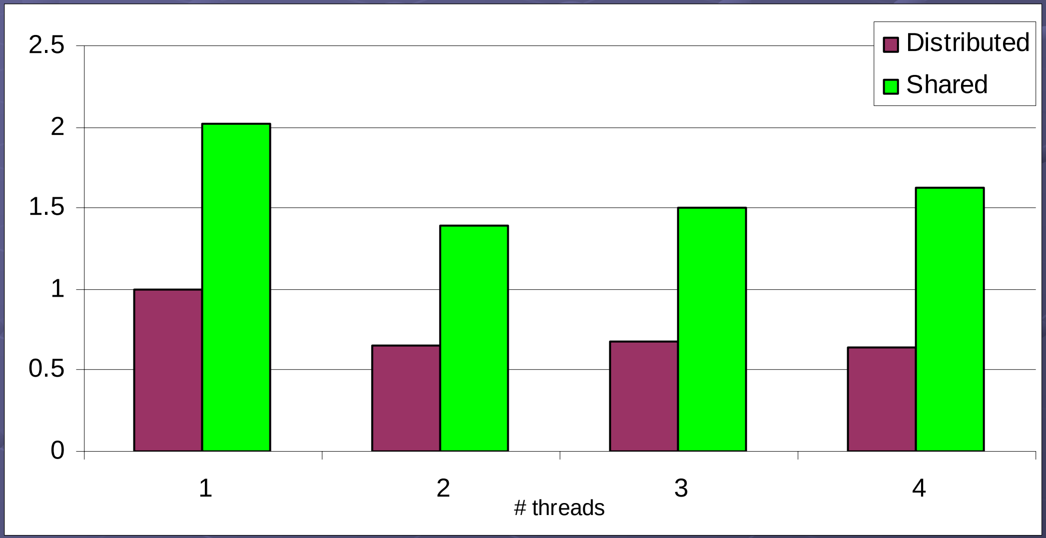
```
for (Iterator iPt = dataPoints.iterator(); iPt.hasNext();) {
    SafePoint point = (SafePoint) iPt.next();
    long dataLock = point.acquireLock();
    if (dataLock != 0) {
        int newCluster = -1;
        double newMinDistance = 10E38;

        for (int iCtr=0; iCtr < clusterCenters.size(); iCtr++) {
            Point center = (Point) clusterCenters.elementAt(iCtr);
            double dist = point.calcDistanceTo(center);
            if (dist < newMinDistance) { // closer center found
                newCluster = iCtr;
                newMinDistance = dist;
            }
        }

        // assign the new cluster center, if it has changed
        if (point.getClusterMembership() != newCluster) {
            point.setClusterMembership(dataLock, newCluster);
            safeChangedMembership.write(true);
        }
        point.releaseLock(dataLock);
    }
}
```

Results

Processing time with respect to single-threaded version

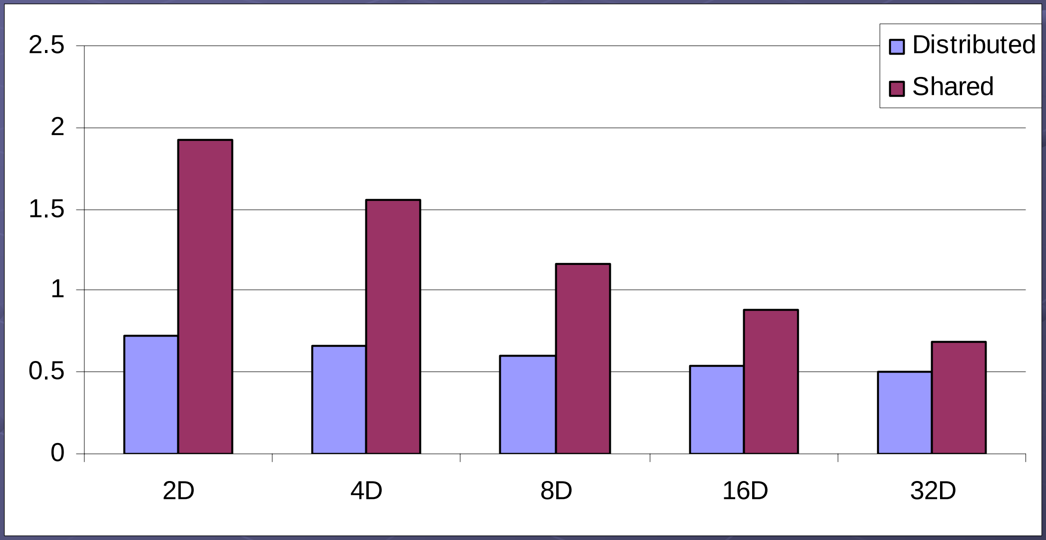


There is a lot of variability in processing time; but, the ratio of processing times is relatively constant.

(on dual core processor)

Results

Processing time with respect to single-threaded version (2 threads)



As the amount of parallelized work increases the ratio of the “locking overhead” to the work done decreases.

(on dual core processor)

Conclusions

- Important to determine if the problem size justifies concurrent implementation.
- Match the number of threads to the number of cores.
- Do not use locks where it is not necessary (pretty high cost). For example, if parts of the code are intended to be run single-threaded, adapt the data structures for such.

Future developments

- Increase the amount of work done in parallel (currently only the cluster assignment is done in parallel; can also put parts of the recalculation of new cluster centers).
- Check other problem characteristics and how they affect the processing time w.r.t. single-threaded version (e.g. total number of points, number of clusters, cluster shape, etc.).
- Compare approach/results to other research.

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?